Platform MPI User's Guide

Platform MPI
Version 8.0
Release date: June 2010
Last modified: June 17, 2010

Platform Computing

Copyright

© 1994-2010 Platform Computing Inc.

Although the information in this document has been carefully reviewed, Platform Computing Corporation ("Platform") does not warrant it to be free of errors or omissions. Platform reserves the right to make corrections, updates, revisions or changes to the information in this document.

UNLESS OTHERWISE EXPRESSLY STATED BY PLATFORM, THE PROGRAM DESCRIBED IN THIS DOCUMENT IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL PLATFORM COMPUTING BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT LIMITATION ANY LOST PROFITS, DATA, OR SAVINGS, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS PROGRAM.

We'd like to hear from you

You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

This document is protected by copyright and you may not redistribute or translate it into another language, in part or in whole.

Internal redistribution

You may only redistribute this document internally within your organization (for example, on an intranet) provided that you continue to check the Platform Web site for updates and update your version of the documentation. You may not make it available to your organization over the Internet.

Trademarks

LSF is a registered trademark of Platform Computing Corporation in the United States and in other jurisdictions.

ACCELERATING INTELLIGENCE, PLATFORM COMPUTING, PLATFORM SYMPHONY, PLATFORM JOB SCHEDULER, PLATFORM ISF, PLATFORM ENTERPRISE GRID ORCHESTRATOR, PLATFORM EGO, and the PLATFORM and PLATFORM LSF logos are trademarks of Platform Computing Corporation in the United States and in other jurisdictions.

UNIX is a registered trademark of The Open Group in the United States and in other jurisdictions.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other products or services mentioned in this document are identified by the trademarks or service marks of their respective owners.

Third-party license agreements

http://www.platform.com/Company/third.part.license.htm

Third-party copyright notices

http://www.platform.com/Company/Third.Party.Copyright.htm

Contents

1 About This Guide 5
- Platforms supported 6
- Documentation resources 10
- Credits 11

2 Introduction 13
- The message passing model 14
- MPI concepts 15

3 Getting Started 27
- Getting started using Linux 28
- Getting started using Windows 35

4 Understanding Platform MPI 49
- Compilation wrapper script utilities 50
- C++ bindings (for Linux) 54
- Autodouble functionality 56
- MPI functions 57
- 64-bit support 58
- Thread-compliant library 59
- CPU binding 60
- MPICH object compatibility for Linux 63
- MPICH2 compatibility 65
- Examples of building on Linux 66
- Running applications on Linux 67
- Running applications on Windows 88
- mpirun options 102
- Runtime environment variables 118
- List of runtime environment variables 121
- Scalability 145
- Dynamic processes 147
- Singleton launching 148
- License release/regain on suspend/resume 149
- Improved deregistration using ptmalloc (Linux only) 150
- Signal propagation (Linux only) 151
- MPI-2 name publishing support 153
- Native language support 154

5 Profiling 155
Using counter instrumentation 156
Using the profiling interface 159
6 Tuning 161
Tunable parameters 162
Message latency and bandwidth 163
Multiple network interfaces 164
Processor subscription 165
Processor locality 166
MPI routine selection 167
7 Debugging and Troubleshooting 169
Debugging Platform MPI applications 170
Troubleshooting Platform MPI applications 174
Appendix A: Example Applications 183
send_receive.f 184
ping_pong.c 185
ping_pong_ring.c (Linux) 187
ping_pong_ring.c (Windows) 191
compute_pi.f 194
master_worker.f90 195
cart.C 196
communicator.c 198
multi_par.f 199
io.c 206
thread_safe.c 207
sort.C 210
compute_pi_spawn.f 216
Appendix B: Large message APIs 219
Appendix C: Standard Flexibility in Platform MPI 229
Platform MPI implementation of standard flexibility 229
Appendix D: mpirun Using Implied prun or srun 231
Implied prun 231
Implied srun 232
Appendix E: Frequently Asked Questions 237
General 237
Installation and setup 238
Building applications 239
Performance problems 240
Network specific 241
Windows specific 242
Appendix F: Glossary 245

CHAPTER 1

About This Guide

This guide describes the Platform MPI implementation of the Message Passing Interface (MPI) standard. This guide helps you use Platform MPI to develop and run parallel applications.

You should have experience developing UNIX applications. You should also understand the basic concepts behind parallel processing, be familiar with MPI, and with the MPI 1.2 and MPI-2 standards (*MPI: A Message-Passing Interface Standard* and *MPI-2: Extensions to the Message-Passing Interface*, respectively).

You can access HTML versions of the MPI 1.2 and 2 standards at *http://www.mpi-forum.org*. This guide supplements the material in the MPI standards and *MPI: The Complete Reference*.

Some sections in this book contain command-line examples to demonstrate Platform MPI concepts. These examples use the /bin/csh syntax.

Platforms supported

Table 1: Supported platforms, interconnects, and operating systems

Platform	Interconnect	Operating System
Intel IA 32	TCP/IP	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	Myrinet GM-2 and MX	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	InfiniBand	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	RDMA Ethernet	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	OFED 1.0, 1.1, 1.2,1.3, 1.4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	uDAPL 1.1, 1.2, 2.0	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5

Platform	Interconnect	Operating System
Intel Itanium-based	TCP/IP	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5, Windows HPCS
	QsNet Elan4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	InfiniBand	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, Cent OS 5, Windows HPCS
	Myrinet GM-2 and MX	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	OFED 1.0, 1.1, 1.2,1.3, 1.4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	uDAPL 1.1, 1.2, 2.0	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
AMD Opteron-based	TCP/IP	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	Myrinet GM-2 and MX	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	InfiniBand	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	QsNet Elan4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	OFED 1.0, 1.1, 1.2,1.3, 1.4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	uDAPL 1.1, 1.2, 2.0	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5

Platform	Interconnect	Operating System
Intel 64	Myrinet GM-2 and MX	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	TCP/IP	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	InfiniBand	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	OFED 1.0, 1.1, 1.2,1.3, 1.4	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	uDAPL 1.1, 1.2, 2.0	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	Red Hat Enterprise Linux AS 4.0 and 5.0, SuSE Linux Enterprise Server 9 and 10, CentOS 5
HP XC3000 Clusters	Myrinet GM-2 and MX	HP XC Linux
	TCP/IP	
	InfiniBand	
	OFED 1.0, 1.1, 1.2,1.3, 1.4	
	uDAPL 1.1, 1.2, 2.0	
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	
HP XC4000 Clusters	QsNet Elan4	HP XC Linux
	Myrinet GM-2 and MX	
	TCP/IP	
	InfiniBand	
	OFED 1.0, 1.1, 1.2,1.3, 1.4	
	uDAPL 1.1, 1.2, 2.0	
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	

Platform	Interconnect	Operating System
HP XC6000 Clusters	TCP/IP	HP XC Linux
	QsNet Elan4	
	InfiniBand	
	OFED 1.0, 1.1, 1.2,1.3, 1.4	
	uDAPL 1.1, 1.2, 2.0	
	QLogic PSM NIC Version: QHT7140, QLE7140 Driver: PSM 1.0, 2.2.1, 2.2	
HP Cluster Platforms	TCP/IP and InfiniBand	Microsoft Windows HPCS 2008

Note:

The last release of HP-MPI for HP-UX was version 2.2.5, which is supported by Platform Computing. This document is for Platform MPI 8.0, which is only being release on Linux and Windows.

Documentation resources

Documentation resources include:

1. Platform MPI product information available at *http://www.platform.com/cluster-computing/platform-mpi*
2. *MPI: The Complete Reference* (2 volume set), MIT Press
3. MPI 1.2 and 2.0 standards available at *http://www.mpi-forum.org*:
    1. *MPI: A Message-Passing Interface Standard*
    2. *MPI-2: Extensions to the Message-Passing Interface*
4. TotalView documents available at *http://www.totalviewtech.com*:
    1. *TotalView Command Line Interface Guide*
    2. *TotalView User's Guide*
    3. *TotalView Installation Guide*
5. Platform MPI release notes available at *http://my.platform.com*.
6. Argonne National Laboratory's implementation of MPI I/O at *http://www-unix.mcs.anl.gov/romio*
7. University of Notre Dame's LAM implementation of MPI at *http://www.lam-mpi.org/*
8. Intel Trace Collector/Analyzer product information (formally known as Vampir) at *http://www.intel.com/software/products/cluster/tcollector/index.htm*and *http://www.intel.com/software/products/cluster/tanalyzer/index.htm*
9. LSF product information at *http://www.platform.com*
10. HP Windows HPC Server 2008 product information at *http://www.microsoft.com/hpc/en/us/product-information.aspx*

Credits

Platform MPI is based on MPICH from Argonne National Laboratory and LAM from the University of Notre Dame and Ohio Supercomputer Center.

Platform MPI includes ROMIO, a portable implementation of MPI I/O developed at the Argonne National Laboratory.

CHAPTER 2

Introduction

The message passing model

Programming models are generally categorized by how memory is used. In the shared memory model, each process accesses a shared address space, but in the message passing model, an application runs as a collection of autonomous processes, each with its own local memory.

In the message passing model, processes communicate with other processes by sending and receiving messages. When data is passed in a message, the sending and receiving processes must work to transfer the data from the local memory of one to the local memory of the other.

Message passing is used widely on parallel computers with distributed memory and on clusters of servers.

The advantages of using message passing include:

- Portability: Message passing is implemented on most parallel platforms.
- Universality: The model makes minimal assumptions about underlying parallel hardware. Message-passing libraries exist on computers linked by networks and on shared and distributed memory multiprocessors.
- Simplicity: The model supports explicit control of memory references for easier debugging.

However, creating message-passing applications can require more effort than letting a parallelizing compiler produce parallel applications.

In 1994, representatives from the computer industry, government labs, and academe developed a standard specification for interfaces to a library of message-passing routines. This standard is known as MPI 1.0 (MPI: A Message-Passing Interface Standard). After this initial standard, versions 1.1 (June 1995), 1.2 (July 1997), and 2.0 (July 1997) have been produced. Versions 1.1 and 1.2 correct errors and minor omissions of MPI 1.0. MPI-2 (MPI-2: Extensions to the Message-Passing Interface) adds new functionality to MPI 1.2. You can find both standards in HTML format at *http://www.mpi-forum.org*.

MPI-1 compliance means compliance with MPI 1.2. MPI-2 compliance means compliance with MPI 2.0. Forward compatibility is preserved in the standard. That is, a valid MPI 1.0 program is a valid MPI 1.2 program and a valid MPI-2 program.

MPI concepts

The primary goals of MPI are efficient communication and portability.

Although several message-passing libraries exist on different systems, MPI is popular for the following reasons:

- Support for full asynchronous communication: Process communication can overlap process computation.
- Group membership: Processes can be grouped based on context.
- Synchronization variables that protect process messaging: When sending and receiving messages, synchronization is enforced by source and destination information, message labeling, and context information.
- Portability: All implementations are based on a published standard that specifies the semantics for usage.

An MPI program consists of a set of processes and a logical communication medium connecting those processes. An MPI process cannot directly access memory in another MPI process. Interprocess communication requires calling MPI routines in both processes. MPI defines a library of routines that MPI processes communicate through.

The MPI library routines provide a set of functions that support the following:

- Point-to-point communications
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Datatype manipulation

Although the MPI library contains a large number of routines, you can design a large number of applications by using the six routines.

Table 2: Six commonly used MPI routines

MPI routine	Description
MPI_Init	Initializes the MPI environment
MPI_Finalize	Terminates the MPI environment
MPI_Comm_rank	Determines the rank of the calling process within a group
MPI_Comm_size	Determines the size of the group
MPI_Send	Sends messages
MPI_Recv	Receives messages

You must call MPI_Finalize in your application to conform to the MPI Standard. Platform MPI issues a warning when a process exits without calling MPI_Finalize.

Caution:

Do not place code before MPI_Init and after MPI_Finalize. Applications that violate this rule are nonportable and can produce incorrect results.

As your application grows in complexity, you can introduce other routines from the library. For example, MPI_Bcast is an often-used routine for sending or broadcasting data from one process to other processes in a single operation.

Use broadcast transfers to get better performance than with point-to-point transfers. The latter use MPI_Send to send data from each sending process and MPI_Recv to receive it at each receiving process.

The following sections briefly introduce the concepts underlying MPI library routines. For more detailed information see MPI: A Message-Passing Interface Standard.

Point-to-point communication

Point-to-point communication involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model and is described in Chapter 3, Point-to-Point Communication in the MPI 1.0 standard.

The performance of point-to-point communication is measured in terms of total transfer time. The total transfer time is defined as

total_transfer_time = *latency* + (*message_size*/*bandwidth*)

where

latency

Specifies the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

message_size

Specifies the size of the message in MB.

bandwidth

Denotes the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in MB per second.

Low latencies and high bandwidths lead to better performance.

Communicators

A communicator is an object that represents a group of processes and their communication medium or context. These processes exchange messages to transfer data. Communicators encapsulate a group of processes so communication is restricted to processes in that group.

The default communicators provided by MPI are *MPI_COMM_WORLD* and *MPI_COMM_SELF*. *MPI_COMM_WORLD* contains all processes that are running when an application begins execution. Each process is the single member of its own *MPI_COMM_SELF* communicator.

Communicators that allow processes in a group to exchange data are termed intracommunicators. Communicators that allow processes in two different groups to exchange data are called intercommunicators.

Many MPI applications depend on knowing the number of processes and the process rank in a given communicator. There are several communication management functions; two of the more widely used are MPI_Comm_size and MPI_Comm_rank.

The process rank is a unique number assigned to each member process from the sequence 0 through (*size*-1), where *size* is the total number of processes in the communicator.

To determine the number of processes in a communicator, use the following syntax:

MPI_Comm_size (MPI_Comm *comm*, int **size*);

where

comm

Represents the communicator handle.

size

Represents the number of processes in the group of *comm*.

To determine the rank of each process in comm, use

MPI_Comm_rank (MPI_Comm *comm*, int **rank*);

where

comm

Represents the communicator handle.

rank

Represents an integer between zero and (*size* - 1).

A communicator is an argument used by all communication routines. The C code example displays the use of MPI_Comm_dup, one of the communicator constructor functions, and MPI_Comm_free, the function that marks a communication object for deallocation.

Sending and receiving messages

There are two methods for sending and receiving data: blocking and nonblocking.

In blocking communications, the sending process does not return until the send buffer is available for reuse.

In nonblocking communications, the sending process returns immediately, and might have started the message transfer operation, but not necessarily completed it. The application might not safely reuse the message buffer after a nonblocking routine returns until MPI_Wait indicates that the message transfer has completed.

In nonblocking communications, the following sequence of events occurs:

1. The sending routine begins the message transfer and returns immediately.
2. The application does some computation.
3. The application calls a completion routine (for example, MPI_Test or MPI_Wait) to test or wait for completion of the send operation.

Blocking communication

Blocking communication consists of four send modes and one receive mode.

The four send modes are:

Standard (MPI_Send)

The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.

Buffered (MPI_Bsend)

The sending process returns when the message is buffered in an application-supplied buffer.

Avoid using the MPI_Bsend mode. It forces an additional copy operation.

Synchronous (MPI_Ssend)

The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.

Ready (MPI_Rsend)

The message is sent as soon as possible.

You can invoke any mode by using the correct routine name and passing the argument list. Arguments are the same for all modes.

For example, to code a standard blocking send, use

MPI_Send (void *_buf_, int _count_, MPI_Datatype _dtype_, int _dest_, int _tag_, MPI_Comm _comm_);

where

buf

Specifies the starting address of the buffer.

count

Indicates the number of buffer elements.

dtype

Denotes the data type of the buffer elements.

dest

Specifies the rank of the destination process in the group associated with the communicator *comm*.

tag

Denotes the message label.

comm

Designates the communication context that identifies a group of processes.

To code a blocking receive, use

MPI_Recv (void *_buf_, int _count_, MPI_datatype _dtype_, int _source_, int _tag_, MPI_Comm _comm_, MPI_Status *_status_);

where

buf

Specifies the starting address of the buffer.

count

Indicates the number of buffer elements.

dtype

Denotes the data type of the buffer elements.

source

Specifies the rank of the source process in the group associated with the communicator *comm*.

tag

Denotes the message label.

comm

Designates the communication context that identifies a group of processes.

status

Returns information about the received message. Status information is useful when wildcards are used or the received message is smaller than expected. Status may also contain error codes.

The `send_receive.f`, `ping_pong.c`, and `master_worker.f90` examples all illustrate the use of standard blocking sends and receives.

Note:

You should not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. Platform MPI sometimes uses buffering for MPI_Send and MPI_Rsend, but it is dependent on message size. Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode.

Nonblocking communication

MPI provides nonblocking counterparts for each of the four blocking send routines and for the receive routine. The following table lists blocking and nonblocking routine calls:

Table 3: MPI blocking and nonblocking calls

Blocking Mode	Nonblocking Mode
MPI_Send	MPI_Isend
MPI_Bsend	MPI_Ibsend
MPI_Ssend	MPI_Issend
MPI_Rsend	MPI_Irsend
MPI_Recv	MPI_Irecv

Nonblocking calls have the same arguments, with the same meaning as their blocking counterparts, plus an additional argument for a request.

To code a standard nonblocking send, use

MPI_Isend(void *_buf_, int _count_, MPI_datatype _dtype_, int_dest_, int _tag_, MPI_Comm _comm_, MPI_Request *_req_);

where

req

Specifies the request used by a completion routine when called by the application to complete the send operation.

To complete nonblocking sends and receives, you can use MPI_Wait or MPI_Test. The completion of a send indicates that the sending process is free to access the send buffer. The completion of a receive indicates that the receive buffer contains the message, the receiving process is free to access it, and the status object that returns information about the received message, is set.

Collective operations

Applications may require coordinated operations among multiple processes. For example, all processes must cooperate to sum sets of numbers distributed among them.

MPI provides a set of collective operations to coordinate operations among processes. These operations are implemented so that all processes call the same operation with the same arguments. Thus, when sending and receiving messages, one collective operation can replace multiple sends and receives, resulting in lower overhead and higher performance.

Collective operations consist of routines for communication, computation, and synchronization. These routines all specify a communicator argument that defines the group of participating processes and the context of the operation.

Collective operations are valid only for intracommunicators. Intercommunicators are not allowed as arguments.

Communication

Collective communication involves the exchange of data among processes in a group. The communication can be one-to-many, many-to-one, or many-to-many.

The single originating process in the one-to-many routines or the single receiving process in the many-to-one routines is called the root.

Collective communications have three basic patterns:

Broadcast and Scatter

Root sends data to all processes, including itself.

Gather

Root receives data from all processes, including itself.

Allgather and Alltoall

Each process communicates with each process, including itself.

The syntax of the MPI collective functions is designed to be consistent with point-to-point communications, but collective functions are more restrictive than point-to-point functions. Important restrictions to keep in mind are:

- The amount of data sent must exactly match the amount of data specified by the receiver.
- Collective functions come in blocking versions only.
- Collective functions do not use a tag argument, meaning that collective calls are matched strictly according to the order of execution.
- Collective functions come in standard mode only.

For detailed discussions of collective communications see Chapter 4, Collective Communication in the MPI 1.0 standard.

The following examples demonstrate the syntax to code two collective operations; a broadcast and a scatter:

To code a broadcast, use

MPI_Bcast(void *_buf_, int _count_, MPI_Datatype _dtype_, int _root_, MPI_Comm _comm_);

where

buf

Specifies the starting address of the buffer.

count

Indicates the number of buffer entries.

dtype

Denotes the datatype of the buffer entries.

root

Specifies the rank of the root.

comm

Designates the communication context that identifies a group of processes.

For example, `compute_pi.f` uses *MPI_BCAST* to broadcast one integer from process 0 to every process in *MPI_COMM_WORLD*.

To code a scatter, use

MPI_Scatter (void* _sendbuf_, int _sendcount_, MPI_Datatype _sendtype_, void* _recvbuf_, int _recvcount_, MPI_Datatype _recvtype_, int _root_, MPI_Comm _comm_);

where

sendbuf

Specifies the starting address of the send buffer.

sendcount

Specifies the number of elements sent to each process.

sendtype

Denotes the datatype of the send buffer.

recvbuf

Specifies the address of the receive buffer.

recvcount

Indicates the number of elements in the receive buffer.

recvtype

Indicates the datatype of the receive buffer elements.

root

Denotes the rank of the sending process.

comm

Designates the communication context that identifies a group of processes.

Computation

Computational operations perform global reduction operations, such as sum, max, min, product, or user-defined functions across members of a group. Global reduction functions include:

Reduce

Returns the result of a reduction at one node.

All-reduce

Returns the result of a reduction at all nodes.

Reduce-Scatter

Combines the functionality of reduce and scatter operations.

Scan

Performs a prefix reduction on data distributed across a group.

Section 4.9, Global Reduction Operations in the MPI 1.0 standard describes each function in detail.

Reduction operations are binary and are only valid on numeric data. Reductions are always associative but might or might not be commutative.

You can select a reduction operation from a defined list (see section 4.9.2 in the MPI 1.0 standard) or you can define your own operation. The operations are invoked by placing the operation name, for example *MPI_SUM* or *MPI_PROD*, in *op*, as described in the MPI_Reducesyntax below.

To implement a reduction, use

MPI_Reduce(void *_sendbuf_, void *_recvbuf_, int _count_, MPI_Datatype _dtype_, MPI_Op _op_, int _root_, MPI_Comm _comm_);

where

sendbuf

Specifies the address of the send buffer.

recvbuf

Denotes the address of the receive buffer.

count

Indicates the number of elements in the send buffer.

dtype

Specifies the datatype of the send and receive buffers.

op

Specifies the reduction operation.

root

Indicates the rank of the root process.

comm

Designates the communication context that identifies a group of processes.

For example `compute_pi.f` uses *MPI_REDUCE* to sum the elements provided in the input buffer of each process in *MPI_COMM_WORLD*, using *MPI_SUM*, and returns the summed value in the output buffer of the root process (in this case, process 0).

Synchronization

Collective routines return as soon as their participation in a communication is complete. However, the return of the calling process does not guarantee that the receiving processes have completed or even started the operation.

To synchronize the execution of processes, call MPI_Barrier. MPI_Barrier blocks the calling process until all processes in the communicator have called it. This is a useful approach for separating two stages of a computation so messages from each stage do not overlap.

To implement a barrier, use

MPI_Barrier(MPI_Comm *comm*);

where

comm

Identifies a group of processes and a communication context.

For example, `cart.C` uses MPI_Barrier to synchronize data before printing.

MPI data types and packing

You can use predefined datatypes (for example, *MPI_INT* in C) to transfer data between two processes using point-to-point communication. This transfer is based on the assumption that the data transferred is stored in contiguous memory (for example, sending an array in a C or Fortran application).

To transfer data that is not homogeneous, such as a structure, or to transfer data that is not contiguous in memory, such as an array section, you can use derived datatypes or packing and unpacking functions:

Derived datatypes

Specifies a sequence of basic datatypes and integer displacements describing the data layout in memory. You can use user-defined datatypes or predefined datatypes in MPI communication functions.

Packing and unpacking functions

Provides MPI_Pack and MPI_Unpack functions so a sending process can pack noncontiguous data into a contiguous buffer and a receiving process can unpack data received in a contiguous buffer and store it in noncontiguous locations.

Using derived datatypes is more efficient than using MPI_Pack and MPI_Unpack. However, derived datatypes cannot handle the case where the data layout varies and is unknown by the receiver (for example, messages that embed their own layout description).

Section 3.12, Derived Datatypes in the MPI 1.0 standard describes the construction and use of derived datatypes. The following is a summary of the types of constructor functions available in MPI:

- Contiguous (MPI_Type_contiguous): Allows replication of a datatype into contiguous locations.
- Vector (MPI_Type_vector): Allows replication of a datatype into locations that consist of equally spaced blocks.
- Indexed (MPI_Type_indexed): Allows replication of a datatype into a sequence of blocks where each block can contain a different number of copies and have a different displacement.
- Structure (MPI_Type_struct): Allows replication of a datatype into a sequence of blocks so each block consists of replications of different datatypes, copies, and displacements.

After you create a derived datatype, you must commit it by calling MPI_Type_commit.

Platform MPI optimizes collection and communication of derived datatypes.

Section 3.13, Pack and unpack in the MPI 1.0 standard describes the details of the pack and unpack functions for MPI. Used together, these routines allow you to transfer heterogeneous data in a single message, thus amortizing the fixed overhead of sending and receiving a message over the transmittal of many elements.

For a discussion of this topic and examples of construction of derived datatypes from the basic datatypes using the MPI constructor functions, see Chapter 3, User-Defined Datatypes and Packing in *MPI: The Complete Reference*.

Multilevel parallelism

By default, processes in an MPI application can only do one task at a time. Such processes are single-threaded processes. This means that each process has an address space with a single program counter, a set of registers, and a stack.

A process with multiple threads has one address space, but each process thread has its own counter, registers, and stack.

Multilevel parallelism refers to MPI processes that have multiple threads. Processes become multithreaded through calls to multithreaded libraries, parallel directives and pragmas, or auto-compiler parallelism.

Multilevel parallelism is beneficial for problems you can decompose into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to do a computation and joins after the computation is complete).

The `multi_par.f` example program is an example of multilevel parallelism.

Advanced topics

This chapter provides a brief introduction to basic MPI concepts. Advanced MPI topics include:

- Error handling
- Process topologies
- User-defined data types

- Process grouping
- Communicator attribute caching
- The MPI profiling interface

To learn more about the basic concepts discussed in this chapter and advanced MPI topics see *MPI: The Complete Reference* and *MPI: A Message-Passing Interface Standard*.

CHAPTER 3

Getting Started

This chapter describes how to get started quickly using Platform MPI. The semantics of building and running a simple MPI program are described, for single and multiple hosts. You learn how to configure your environment before running your program. You become familiar with the file structure in your Platform MPI directory. The Platform MPI licensing policy is explained.

The goal of this chapter is to demonstrate the basics to getting started using Platform MPI. It is separated into two major sections: *Getting Started Using Linux*, and *Getting Started Using Windows*.

Getting started using Linux

Configuring your environment

Setting *PATH*

If you move the Platform MPI installation directory from its default location in `/opt/platform_mpi` for Linux:

- Set the *MPI_ROOT* environment variable to point to the location where MPI is installed.
- Add `$MPI_ROOT/bin` to *PATH*.
- Add `$MPI_ROOT/share/man` to *MANPATH*.

MPI must be installed in the same directory on every execution host.

Setting up remote shell

By default, Platform MPI attempts to use `ssh` on Linux. We recommend that `ssh` users set `StrictHostKeyChecking=no` in their `~/.ssh/config`.

To use a different command such as "`rsh`" for remote shells, set the *MPI_REMSH* environment variable to the desired command. The variable is used by `mpirun` when launching jobs, as well as by the `mpijob` and `mpiclean` utilities. Set it directly in the environment by using a command such as:

```
% setenv MPI_REMSH "ssh -x
```

The tool specified with *MPI_REMSH* must support a command-line interface similar to the standard utilities `rsh`, `remsh`, and `ssh`. The -n option is one of the arguments `mpirun` passes to the remote shell command.

If the remote shell does not support the command-line syntax Platform MPI uses, write a wrapper script such as `/path/to/myremsh` to change the arguments and set the *MPI_REMSH* variable to that script.

Platform MPI supports setting *MPI_REMSH* using the -e option to `mpirun`:

```
% $MPI_ROOT/bin/mpirun -e MPI_REMSH=ssh <options> -f \ <appfile>
```

Platform MPI also supports setting *MPI_REMSH* to a command that includes additional arguments (for example "ssh -x". But, if this is passed to `mpirun` with -e MPI_REMSH= then the parser in Platform MPI V2.2.5.1 requires additional quoting for the value to be correctly received by `mpirun`:

```
% $MPI_ROOT/bin/mpirun -e MPI_REMSH="ssh -x" <options> -f <appfile>
```

When using `ssh`, be sure it is possible to use `ssh` from the host where `mpirun` is executed to other nodes without `ssh` requiring interaction from the user. Also ensure `ssh` functions between the worker-nodes because the `ssh` calls used to launch the job are not necessarily all started by `mpirun` directly (a tree of `ssh` calls is used for improved scalability).

Compiling and running your first application

To quickly become familiar with compiling and running Platform MPI programs, start with the C version of a familiar `hello_world` program. The source file for this program is called `hello_world.c`. The program prints out the text string "Hello world! I'm *r* of *s* on *host*" where *r* is a process's rank, *s* is the size of the communicator, and *host* is the host where the program is run. The processor name is the host name for this implementation. The processor name is the host name for this implementation. Platform MPI returns the host name for *MPI_Get_processor_name*.

The source code for `hello_world.c` is stored in `$MPI_ROOT/help` and is shown below.

```
#include <stdio.h>
#include "mpi.h"
void main(argc, argv)
int argc;
char *argv[];
{
   int rank, size, len;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Get_processor_name(name, &len);
   printf("Hello world!I'm %d of %d on %s\n", rank, size, name);
   MPI_Finalize();
   exit(0);
}
```

Building and running on a single host

This example teaches you the basic compilation and run steps to execute `hello_world.c` on your local host with four-way parallelism. To build and run `hello_world.c` on a local host named `jawbone`:

1. Change to a writable directory.
2. Compile the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**
3. Run the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpirun -np 4 hello_world**

   where

   -np 4 specifies 4 as the number of rocesses to run.
4. Analyze `hello_world` output.

   Platform MPI prints the output from running the `hello_world` executable in nondeterministic order. The following is an example of the output:

   ```
   Hello world! I'm 1 of 4 on jawbone
   Hello world! I'm 3 of 4 on jawbone
   Hello world! I'm 0 of 4 on jawbone
   Hello world! I'm 2 of 4 on jawbone
   ```

Building and running on a Linux cluster using appfiles

The following is an example of basic compilation and run steps to execute `hello_world.c` on a cluster with 4-way parallelism. To build and run `hello_world.c` on a cluster using an appfile:

1. Change to a writable directory.
2. Compile the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**
3. Create the file `appfile` for running on nodes n01 and n02 as:

   ```
   -h n01 -np 2 /path/to/hello_world -h n02 -np 2 /path/to/hello_world
   ```
4. Run the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpirun -f *appfile***

By default, `mpirun` will `rsh`/`remsh` to the remote machines n01 and n02. If desired, the environment variable *MPI_REMSH* can be used to specify a different command, such as `/usr/bin/ssh` or `"ssh -x"`.

5. Analyze `hello_world` output.

   Platform MPI prints the output from running the `hello_world` executable in nondeterministic order. The following is an example of the output:

```
Hello world! I'm 1 of 4 n01
Hello world! I'm 3 of 4 n02
Hello world! I'm 0 of 4 n01
Hello world! I'm 2 of 4 n02
```

Building and running on an HP XC cluster using srun

The following is an example of basic compilation and run steps to execute hello_world.c on an HP XC cluster with 4-way parallelism. To build and run hello_world.c on an HP XC cluster (assuming LSF is not installed):

1. Change to a writable directory.
2. Compile the hello_world executable file:

   % **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**

3. Run the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpirun -srun -n4 hello_world**

   where

   -n4 specifies 4 as the number of processes to run from SLURM.

4. Analyze `hello_world` output.

   Platform MPI prints the output from running the hello_world executable in nondeterministic order. The following is an example of the output:

```
I'm 1 of 4 n01 Hello world!
I'm 3 of 4 n02 Hello world!
I'm 0 of 4 n01 Hello world!
I'm 2 of 4 n02 Hello world!
```

Directory structure for Linux

Platform MPI files are stored in the `/opt/platform_mpi` directory for Linux.

If you move the Platform MPI installation directory from its default location in `/opt/platform_mpi`, set the *MPI_ROOT* environment variable to point to the new location. The directory structure is organized as follows:

Table 4: Directory structure for Linux

Subdirectory	Contents
bin	Command files for the Platform MPI utilities gather_info script
etc	Configuration files (for example, `pmpi.conf`)
help	Source files for the example programs
include	Header files

Subdirectory	Contents
lib/pa2.0	Platform MPI PA-RISC 32-bit libraries
lib/pa20_64	Platform MPI PA-RISC 64-bit libraries
lib/linux_ia32	Platform MPI Linux 32-bit libraries
lib/linux_ia64	Platform MPI Linux 64-bit libraries for Itanium
lib/linux_amd64	Platform MPI Linux 64-bit libraries for Opteron and Intel64
modules	OS kernel module files
MPICH1.2/	MPICH 1.2 compatibility wrapper libraries
MPICH2.0/	MPICH 2.0 compatibility wrapper libraries
newconfig/	Configuration files and release notes
sbin	Internal Platform MPI utilities
share/man/man1*	manpages for Platform MPI utilities
share/man/man3*	manpages for Platform MPI library
doc	Release notes
licenses	License files

Linux man pages

The manpages are in the `$MPI_ROOT/share/man/man1*` subdirectory for Linux. They can be grouped into three categories: general, compilation, and run-time. One general manpage, MPI.1 is an overview describing general features of Platform MPI. The compilation and run-time manpages describe Platform MPI utilities.

The following table describes the three categories of manpages in the man1 subdirectory that comprise manpages for Platform MPI utilities:

Table 5: Linux man page categories

Category	manpages	Description
General	MPI.1	Describes the general features of Platform MPI.
Compilation	• mpicc.1 • mpiCC.1 • mpif77.1 • mpif90.1	Describes the available compilation utilities.

Category	manpages	Description
Runtime	• mpiclean.1 • mpidebug.1 • mpienv.1 • mpiexec.1 • mpijob.1 • mpimtsafe.1 • mpirun.1 • mpistdio.1 • autodbl.1	Describes run-time utilities, environment variables, debugging, thread-safe, and diagnostic libraries.

Licensing policy for Linux

Platform MPI for Linux uses FlexNet Publisher (formerly FLEXlm) licensing technology. A license file can be named `license.dat` or any file name with an extension of `.lic`. The license file must be placed in the installation directory (default location `/opt/pcmpi/licenses`) on all run-time systems.

Platform MPI has an Independent Software Vendor (ISV) program that allows participating ISVs to freely distribute Platform MPI with their applications. When the application is part of the Platform MPI ISV program, there is no licensing requirement for the user. The ISV provides a licensed copy of Platform MPI. Contact your application vendor to find out if they participate in the Platform MPI ISV program. The copy of Platform MPI distributed with a participating ISV works with that application. A Platform MPI license is required for all other applications.

Licensing for Linux

Platform MPI now supports redundant license servers using the FLEXnet Publisher licensing software. Three servers can create a redundant license server network. For a license checkout request to be granted, at least two servers must be running and able to communicate with each other. This avoids a single-license server failure which would prevent new Platform MPI jobs from starting. With three-server redundant licensing, the full number of Platform MPI licenses can be used by a single job.

When selecting redundant license servers, use stable nodes that are not rebooted or shutdown frequently. The redundant license servers exchange heartbeats. Disruptions to that communication can cause the license servers to stop serving licenses.

The redundant license servers must be on the same subnet as the Platform MPI compute nodes. They do not have to be running the same version of operating system as the Platform MPI compute nodes, but it is recommended. Each server in the redundant network must be listed in the Platform MPI license key by *hostname* and *hostid*. The *hostid* is the MAC address of the eth0 network interface. The eth0 MAC address is used even if that network interface is not configured.

The *hostid* can be obtained by typing the following command if Platform MPI is installed on the system:

```
% /opt/platform_mpi/bin/licensing/arch/lmutil lmhostid
```

The eth0 MAC address can be found using the following command:

```
% /sbin/ifconfig | egrep "^eth0" | awk '{print $5}' | sed s/://g
```

The *hostname* can be obtained by entering the command `hostname`.To request a three server redundant license key for Platform MPI 7.1 for Linux, contact Platform Computing. For more information, see your license certificate.

Installing license files

A valid license file contains the system host ID and the associated license key. License files can be named as `license.dat` or any name with extension of `*.lic` (for example, `mpi.lic`). Copy the license file under the directory `/opt/platform_mpi/licenses`.

The command to run the license server is:

% **$MPI_ROOT/bin/licensing/<*arch*>/lmgrd -c mpi.lic**

License testing

To check for a license, build and run the hello_world program in `$MPI_ROOT/help/hello_world.c`. If your system is not properly licensed, you receive the following error message: `(MPI BUG: Valid MPI license not found in search path)`

Merging licenses

Newer Platform MPI licenses use the INCREMENT feature which allows separate Platform MPI licenses to be used in combination by concatenating files. For example:

License 1:

```
SERVER myserver 0014c2c1f34a DAEMON HPQ INCREMENT platform_mpi lsf_ld 1.0
permanent 8 9A40ECDE2A38 \                    NOTICE="License Number =
AAAABBBB1111" SIGN=E5CEDE3E5626
```

License 2:

```
SERVER myserver 0014c2c1f34a DAEMON HPQ INCREMENT platform_mpi lsf_ld 1.0
permanent 16 BE468B74B592 \                    NOTICE="License Number =
AAAABBBB2222" SIGN=9AB4034C6CB2
```

Here, License 1 is for 8 ranks, and License 2 is for 16 ranks. The two licenses can be combined into a single file:

```
SERVER myserver 0014c2c1f34a DAEMON HPQ INCREMENT platform_mpi lsf_ld 1.0
permanent 8 9A40ECDE2A38 \            NOTICE="License Number = AAAABBBB1111"
SIGN=E5CEDE3E5626

SERVER myserver 0014c2c1f34a DAEMON HPQ INCREMENT platform_mpi lsf_ld 1.0
permanent 16 BE468B74B592 \            NOTICE="License Number = AAAABBBB2222"
SIGN=9AB4034C6CB2
```

The result is a valid license for 24 ranks.

Redundant license servers

Platform MPI supports redundant license servers using the FLEXnet Publisher licensing software. Three servers can create a redundant license server network. For a license checkout request to be granted, at least two servers must be running and able to communicate with each other. This avoids a single-license server failure which would prevent new Platform MPI jobs from starting. With three-server redundant licensing, the full number of Platform MPI licenses can be used by a single job.

When selecting redundant license servers, use stable nodes that are not rebooted or shutdown frequently. The redundant license servers exchange heartbeats. Disruptions to that communication can cause the license servers to stop serving licenses.

The redundant license servers must be on the same subnet as the Platform MPI compute nodes. They do not have to be running the same version of operating system as the Platform MPI compute nodes, but it is recommended.

Each server in the redundant network must be listed in the Platform MPI license key by hostname and hostid. The hostid is the MAC address of the eth0 network interface. The eth0 MAC address is used even if that network interface is not configured. The hostid can be obtained by typing the following command if Platform MPI is installed on the system:

```
% /opt/platform_mpi/bin/licensing/arch/lmutil lmhostid
```

The eth0 MAC address can be found using the following command:

```
% /sbin/ifconfig | egrep "^eth0" | awk '{print $5}' | sed s/://g
```

The hostname can be obtained by entering the command `hostname`.

To request a three server redundant license key for Platform MPI for Linux, contact the Platform Computing.

Version identification

To determine the version of a Platform MPI installation, use the `ident` or `rpm` command on Linux.

For example:

```
% mpirun -version
```

or

```
% rpm -qa | grep platform_mpi
```

Getting started using Windows

Configuring your environment

The default install directory location for Platform MPI for Windows is one of the following directories:

On 64-bit Windows:

C:\Program Files (x86)\Platform Computing\Platform-MPI

On 32-bit Windows:

C:\Program Files\Platform Computing\Platform-MPI

The default install defines the system environment variable *MPI_ROOT*, but does not put "`%MPI_ROOT%\bin`" in the system path or your user path.

If you choose to move the Platform MPI installation directory from its default location:

1. Change the system environment variable *MPI_ROOT* to reflect the new location.
2. You may need to add "`%MPI_ROOT%\bin\mpirun.exe`", "`%MPI_ROOT%\bin\mpid.exe`", "`%MPI_ROOT%\bin\mpidiag.exe`", and "`%MPI_ROOT%\bin\mpisrvutil.exe`" to the firewall exceptions depending on how your system is configured.

Platform MPI must be installed in the same directory on every execution host.

To determine the version of a Platform MPI installation, use the -version flag with the `mpirun` command:

```
C:\>  "%MPI_ROOT%\bin\mpirun" -version
```

Setting environment variables

Environment variables can be used to control and customize the behavior of a Platform MPI application. The environment variables that affect the behavior of Platform MPI at run time are described in the mpienv(1) manpage.

In all run modes, Platform MPI enables environment variables to be set on the command line with the -e option. For example:

```
C:\> "%MPI_ROOT%\bin\mpirun" -e MPI_FLAGS=y40 -f appfile
```

See the *Platform MPI User's Guide* for more information on setting environment variables globally using the command line.

On Windows 2008, environment variables can be set from the GUI or on the command line.

From the GUI, select New Job > Task List (from the left menu list) and select an existing task. Set the environment variable in the Task Properties window at the bottom.

Note:

Set these environment variables on the `mpirun` task.

Environment variables can also be set using the flag /env. For example:

```
C:\> job add JOBID /numprocessors:1 /env:"MPI_ROOT=\\shared\alternate\location" ...
```

Compiling and running your first application

To quickly become familiar with compiling and running Platform MPI programs, start with the C version of the familiar hello_world program. This program is called hello_world.c and prints out the text string "Hello world! I'm *r* of *s* on *host*" where *r* is a process's rank, *s* is the size of the communicator, and *host* is the host where the program is run.

The source code for hello_world.c is stored in `%MPI_ROOT%\help`.

Command-line basics

The utility `"%MPI_ROOT%\bin\mpicc"` is included to aid in command line compilation. To compile with this utility, set *MPI_CC* to the path of the command line compiler you want to use. Specify -mpi32 or -mpi64 to indicate if you are compiling a 32- or 64-bit application. Specify the command line options that you normally pass to the compiler on the `mpicc` command line. The `mpicc` utility adds additional command line options for Platform MPI include directories and libraries. The -show option can be specified to `mpicc` to display the command generated without executing the compilation command. See the manpage mpicc(1) for more information.

To construct the desired compilation command, the `mpicc` utility needs to know what command line compiler is to be used, the bitness of the executable that compiler will produce, and the syntax accepted by the compiler. These can be controlled by environment variables or from the command line.

Table 6: mpicc Utility

Environment Variable	Value	Command Line
MPI_CC	desired compiler (default cl)	-mpicc <*value*>
MPI_BITNESS	32 or 64 (no default)	-mpi32 or -mpi64
MPI_WRAPPER_SYNTAX	windows or unix (default windows)	-mpisyntax <*value*>

For example, to compile `hello_world.c` using a 64-bit 'cl' contained in your `PATH` could be done with the following command since 'cl' and the 'Windows' syntax are defaults:

`C:\>` **"%MPI_ROOT%\bin\mpicc" -mpi64 hello_world.c /link /out:hello_world_cl64.exe**

Or, use the following example to compile using the PGI compiler which uses a more UNIX-like syntax:

`C:\>` **"%MPI_ROOT%\bin\mpicc" -mpicc pgcc -mpisyntax unix -mpi32 hello_world.c -o hello_world_pgi32.exe**

To compile C code and link against Platform MPI without utilizing the `mpicc` tool, start a command prompt that has the appropriate environment settings loaded for your compiler, and use it with the compiler option:

/I"%MPI_ROOT%\include<*32|64*>"

and the linker options:

/libpath:"%MPI_ROOT%\lib" /subsystem:console <*libhpmpi64.lib|libhpmpi32.lib*>

The above assumes the environment variable *MPI_ROOT* is set.

For example, to compile `hello_world.c` from the Help directory using Visual Studio (from a Visual Studio 2008 command prompt window):

cl hello_world.c /I"%MPI_ROOT%\include\64" /link /out:hello_world.exe /libpath:"%MPI_ROOT%\lib" /subsystem:console libhpmpi64.lib

The PGI compiler uses a more UNIX-like syntax. From a PGI command prompt:

pgcc hello_world.c -I"%MPI_ROOT%\include\64" -o hello_world.exe -L"%MPI_ROOT%\lib" -lhpmpi64

mpicc.bat

The `mpicc.bat` script links by default using the static run-time libraries /MT. This behavior allows the application to be copied without any side effects or additional link steps to embed the manifest library.

When linking with /MD (dynamic libraries), you must copy the generated <*filename*>`.exe.manifest` along with the `.exe/.dll` file or the following run-time error will display:

```
This application has failed to start because MSVCR90.dll was not found. Re-installing the application may fix this problem.
```

To embed the manifest file into `.exe/.dll`, use the `mt` tool. For more information, see the Microsoft/ Visual Studio `mt.exe` tool.

The following example shows how to embed a `.manifest` file into an application:

`C:\>` **"%MPI_ROOT%\bin\mpicc.bat" -mpi64 /MD hello_world.c**

C:\> **mt -manifest hello_world.exe.manifest -outputresource:hello_world.exe;1**

Fortran command-line basics

The utility `"%MPI_ROOT%\bin\mpif90"` is included to aid in command line compilation. To compile with this utility, set *MPI_F90* to the path of the command line compiler you want to use. Specify -mpi32 or -mpi64 to indicate if you are compiling a 32- or 64-bit application. Specify the command line options that you normally pass to the compiler on the `mpif90` command line. The `mpif90` utility adds additional command line options for Platform MPI include directories and libraries. The -show option can be specified to `mpif90` to display the command generated without executing the compilation command. See the manpage mpif90(1) for more information.

To construct the desired compilation command, the `mpif90` utility needs to know what command line compiler is to be used, the bitness of the executable that compiler will produce, and the syntax accepted by the compiler. These can be controlled by environment variables or from the command line.

Table 7: mpif90 utility

Environment Variable	Value	Command Line
MPI_F90	desired compiler (default ifort)	-mpif90 <*value*>
MPI_BITNESS	32 or 64 (no default)	-mpi32 or -mpi64
MPI_WRAPPER_SYNTAX	windows or unix (default windows)	-mpisyntax <*value*>

For example, to compile `compute_pi.f` using a 64-bit 'ifort' contained in your `PATH` could be done with the following command since 'ifort' and the 'Windows' syntax are defaults:

`C:\>` **"%MPI_ROOT%\bin\mpif90" -mpi64 compute_pi.f /link /out:compute_pi_ifort.exe**

Or, use the following example to compile using the PGI compiler which uses a more UNIX-like syntax:

`C:\>` **"%MPI_ROOT%\bin\mpif90" -mpif90 pgf90 -mpisyntax unix -mpi32 compute_pi.f -o compute_pi_pgi32.exe**

To compile `compute_pi.f` using Intel Fortran without utilizing the `mpif90` tool (from a command prompt that has the appropriate environment settings loaded for your Fortran compiler):

```
C:\> ifort compute_pi.f /I"%MPI_ROOT%\include\64" /link /out:compute_pi.exe /libpath:"%MPI_ROOT%\lib" /subsystem:console libhpmpi64.lib
```

Note:

Intel compilers often link against the Intel run-time libraries. When running an MPI application built with the Intel Fortran or C/C++ compilers, you might need to install the Intel run-time libraries on every node of your cluster. We recommend that you install the version of the Intel run-time libraries that correspond to the version of the compiler used on the MPI application.

Autodouble (automatic promotion)

Platform MPI supports automatic promotion of Fortran datatypes using any of the following arguments (some of which are not supported on all Fortran compilers).

1. /integer_size:64
2. /4I8
3. -i8
4. /real_size:64
5. /4R8
6. /Qautodouble
7. -r8

If these flags are given to the `mpif90.bat` script at link time, then the application will be linked enabling Platform MPI to interpret the datatype *MPI_REAL* as 8 bytes (etc. as appropriate) at runtime.

However, if your application is written to explicitly handle the autodoubled datatypes (for example, if a variable is declared real and the code is compiled -r8 and corresponding MPI calls are given *MPI_DOUBLE* for the datatype), then the autodouble related command line arguments should not be passed to `mpif90.bat` at link time (because that would cause the datatypes to be automatically changed).

Building and running on a single host

The following example describes the basic compilation and run steps to execute `hello_world.c` on your local host with 4-way parallelism. To build and run `hello_world.c` on a local host named banach1:

1. Change to a writable directory, and copy `hello_world.c` from the help directory:

   ```
   C:\>  copy "%MPI_ROOT%\help\hello_world.c" .
   ```

2. Compile the hello_world executable file.

   In a proper compiler command window (for example, Visual Studio command window), use `mpicc` to compile your program:

   ```
   C:\>  "%MPI_ROOT%\bin\mpicc" -mpi64 hello_world.c
   ```

   Note:

   Specify the bitness using -mpi64 or -mpi32 for `mpicc` to link in the correct libraries. Verify you are in the correct 'bitness' compiler window. Using -mpi64 in a Visual Studio 32-bit command window does not work.

3. Run the hello_world executable file:

   ```
   C:\>  "%MPI_ROOT%\bin\mpirun" -np 4 hello_world.exe
   ```

   where -np 4 specifies 4 as the number of processors to run.
4. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

   ```
   Hello world! I'm 1 of 4 on banach1
   Hello world! I'm 3 of 4 on banach1
   Hello world! I'm 0 of 4 on banach1
   Hello world! I'm 2 of 4 on banach1
   ```

Building and running multihost on Windows HPCS clusters

The following is an example of basic compilation and run steps to execute `hello_world.c` on a cluster with 16-way parallelism. To build and run `hello_world.c` on a HPCS cluster:

1. Change to a writable directory on a mapped drive. Share the mapped drive to a folder for the cluster.
2. Open a Visual Studio command window. (This example uses a 64-bit version, so a Visual Studio 64-bit command window is opened.)
3. Compile the hello_world executable file:

   ```
   X:\demo> set MPI_CC=cl
   ```

   ```
   X:\demo> "%MPI_ROOT%\bin\mpicc" -mpi64 "%MPI_ROOT%\help\hello_world.c"
   ```

   ```
   Microsoft C/C++ Optimizing Compiler Version 14.00.50727.42 for 64-bit
   Copyright Microsoft Corporation. All rights reserved.
   hello_world.c
   Microsoft Incremental Linker Version 8.00.50727.42
   Copyright Microsoft Corporation. All rights reserved.
   /out:hello_world.exe
   "/libpath:C:\Program Files (x86)\Platform-MPI\lib"
   /subsystem:console
   libpcmpi64.lib
   libmpio64.lib
   hello_world.obj
   ```
4. Create a new job requesting the number of CPUs to use. Resources are not yet allocated, but the job is given a JOBID number which is printed to stdout:

   ```
   C:\>  job new /numprocessors:16 /exclusive:true
   ```

   ```
   Job queued, ID: 4288
   ```
5. Add a single-CPU `mpirun` task to the newly created job. The `mpirun` job creates more tasks filling the rest of the resources with the compute ranks, resulting in a total of 16 compute ranks for this example:

   ```
   C:\>  job add 4288 /numprocessors:1 /exclusive:true /stdout:\\ node\path\to\a\shared\file .out /
   stderr:\\ node\path\to\a\shared\file .err "%MPI_ROOT%\bin\mpirun" -hpc \\ node\path\to
   \hello_world.exe
   ```
6. Submit the job. The machine resources are allocated and the job is run.

   ```
   C:\>  job submit /id:4288
   ```

Multiple-Program Multiple-Data (MPMD)

To run Multiple-Program Multiple-Data (MPMD) applications or other more complex configurations that require further control over the application layout or environment, dynamically create an appfile

within the job using the utility "%MPI_ROOT%\bin\mpi_nodes.exe" as in the following example. To create the executable, perform Steps 1 through 3 from the previous section. Then continue with:

1. Create a new job.

   C:\> **job new /numprocessors:16 /exclusive:true**

   ```
   Job queued, ID: 4288
   ```

2. Submit a script. Verify *MPI_ROOT* is set in the environment (See the mpirun manpage for more information):

   C:\> **job add 4288 /numprocessors:1 /env:MPI_ROOT="%MPI_ROOT%" /exclusive:true /stdout:** *node\path\to\a\shared\file* **.out /stderr:** *node\path\to\a\shared\file* **.err** *path\submission_script* **.vbs**

   Where submission_script.vbs contains code such as:

   ```
   Option Explicit
   Dim sh, oJob, JobNewOut, appfile, Rsrc, I, fs
   Set sh = WScript.CreateObject(WScript.Shell)
   Set fs = CreateObject(Scripting.FileSystemObject)
   Set oJob = sh.exec(%MPI_ROOT%\bin\mpi_nodes.exe)
   JobNewOut = oJob.StdOut.Readall
   Set appfile = fs.CreateTextFile(<path>\appfile, True)
   Rsrc = Split(JobNewOut, )
   For I = LBound(Rsrc) + 1 to UBound(Rsrc) Step 2
          appfile.WriteLine(-h + Rsrc(I) + -np + Rsrc(I+1) + _
                   <path>\foo.exe )
   Next
   appfile.Close
   Set oJob = sh.exec(%MPI_ROOT%\bin\mpirun.exe -TCP -f _
          <path>\appfile )
   wscript.Echo oJob.StdOut.Readall
   ```

3. Submit the job as in the previous example:

   C:\> **job submit /id:4288**

The above example using submission_script.vbs is only an example. Other scripting languages can be used to convert the output of mpi_nodes.exe into an appropriate appfile.

Building and running multihost on Windows HPCS clusters

The following is an example of basic compilation and run steps to execute hello_world.c on a cluster with 16-way parallelism. To build and run hello_world.c on an HPCS cluster:

1. Change to a writable directory on a mapped drive. The mapped drive should be to a shared folder for the cluster.
2. Open a Visual Studio command window. (This example uses a 64-bit version, so a Visual Studio x64 command window opens.)
3. Compile the hello_world executable file:

   X:\Demo> **"%MPI_ROOT%\bin\mpicc" -mpi64 "%MPI_ROOT%\help\hello_world.c"**

   ```
   Microsoft C/C++ Optimizing Compiler Version 14.00.50727.42 for x64
    Copyright Microsoft Corporation. All rights reserved.

    hello_world.c
    Microsoft Incremental Linker Version 8.00.50727.42
    Copyright Microsoft Corporation. All rights reserved.

     /out:hello_world.exe
     "/libpath:C:\Program Files (x86)\Platform-MPI\lib"
    /subsystem:console
   ```

```
libpcmpi64.lib
libmpio64.lib
hello_world.obj
```

4. Create a job requesting the number of CPUs to use. Resources are not yet allocated, but the job is given a JOBID number that is printed to stdout:

   > **job new /numprocessors:16**

   ```
   Job queued, ID: 4288
   ```

5. Add a single-CPU `mpirun` task to the newly created job. `mpirun` creates more tasks filling the rest of the resources with the compute ranks, resulting in a total of 16 compute ranks for this example:

   > **job add 4288 /numprocessors:1 /stdout:*node**path**to*\a\s*hared**file*.out /stderr:*node**path**to*\a\s*hared**file*.err "%MPI_ROOT%\bin\mpirun" -ccp *node**path**to*\hello_world.exe**

6. Submit the job. The machine resources are allocated and the job is run.

   > **job submit /id:4288**

Building and running MPMD applications on Windows HPCS

To run Multiple-Program Multiple-Data (MPMD) applications or other more complex configurations that require further control over the application layout or environment, use an appfile to submit the Platform MPI job through the HPCS scheduler.

Create the appfile indicating the node for the ranks using the -h <*node* flag and the rank count for the given node using the -n X flag. Ranks are laid out in the order they appear in the appfile. Submit the job using `mpirun -ccp -f <`*appfile*>. For this example, the hello_world.c program is copied to simulate a server and client program in an MPMD application. The print statement for each is modified to indicate `server` or `client` program so the MPMD application can be demonstrated:

1. Change to a writable directory on a mapped drive. The mapped drive should be to a shared folder for the cluster.
2. Open a Visual Studio command window. This example uses a 64-bit version, so a Visual Studio x64 command window is opened.
3. Copy the hello_world.c source to server.c and client.c. Then edit each file to change the print statement and include server and client in each:

   `X:\Demo>` **copy "%MPI_ROOT\help\hello_world.c" .\server.c**

   `X:\Demo>` **copy "%MPI_ROOT\help\hello_world.c" .\server.c**

   Edit each to modify the print statement for both .c files to include server or client in the print so the executable being run is visible.

4. Compile the server.c and client.c programs:

   `X:\Demo>` **"%MPI_ROOT%\bin\mpicc" /mpi64 server.c**

```
Microsoft (R) C/C++ Optimizing Compiler Version 14.00.50727.762 for x64
Copyright (C) Microsoft Corporation. All rights reserved.
server.c
Microsoft (R) Incremental Linker Version 8.00.50727.762
Copyright (C) Microsoft Corporation. All rights reserved.
 /out:server.exe
"/libpath:C:\Program Files (x86)\Platform-MPI\lib"
 /subsystem:console
libhpcmpi64.lib
libmpio64.lib
server.obj
```

```
X:\Demo>  "%MPI_ROOT%\bin\mpicc" /mpi64 client.c
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 14.00.50727.762 for x64
Copyright (C) Microsoft Corporation. All rights reserved.client.c
Microsoft (R) Incremental Linker Version 8.00.50727.762
Copyright (C) Microsoft Corporation. All rights reserved./out:client.exe
"/libpath:C:\Program Files (x86)\Platform-MPI\lib"
/subsystem:console
libpcmpi64.lib
libmpio64.lib
client.obj
```

5. Create an appfile that uses your executables.

   For example, create the following appfile, `appfile.txt`:

   ```
   -np 1 -h node1 server.exe
   -np 1 -h node1 client.exe
   -np 2 -h node2 client.exe
   -np 2 -h node3 client.exe
   ```

   This appfile runs one server rank on node1, and 5 client ranks: one on node1, two on node2, and two on node3.

6. Submit the job using appfile mode:

   ```
   X:\work> "%MPI_ROOT%\bin\mpirun" -ccp -f appfile.txt
   ```

   This submits the job to the scheduler, allocating the nodes indicated in the appfile. Output and error files defaults to `appfile-<JOBID>-<TASKID>.out` and `appfile-<JOBID>-<TASKID>.err` respectively. These file names can be altered using the -ccpout and -ccperr flags.

7. Check your results. Assuming the job submitted was job ID 98, the file `appfile-98.1.out` was created. The file content is:

   ```
   X:\Demo>  type appfile-98.1.out
   ```

   ```
   Hello world (Client)! I'm 2 of 6 on node2
   Hello world (Client)! I'm 1 of 6 on node1
   Hello world (Server)! I'm 0 of 6 on node1
   Hello world (Client)! I'm 4 of 6 on node3
   Hello world (Client)! I'm 5 of 6 on node3
   Hello world (Client)! I'm 3 of 6 on node2
   ```

Building an MPI application on Windows with Visual Studio and using the property pages

To build an MPI application on Windows in C or C++ with VS2008, use the property pages provided by Platform MPI to help link applications.

Two pages are included with Platform MPI and are located at the installation location (*MPI_ROOT*) in `help\PMPI.vsprops` and `PMPI64.vsprops`.

Go to VS Project > View > Property Manager. Expand the project. This shows configurations and platforms set up for builds. Include the correct property page (`PMPI.vsprops` for 32-bit apps, `PMPI64.vsprops` for 64-bit apps) in the Configuration/Platform section.

Select this page by double-clicking the page or by right-clicking on the page and selecting Properties. Go to the User Macros section. Set *MPI_ROOT* to the desired location (i.e., the installation location of Platform MPI). This should be set to the default installation location:

```
% ProgramFiles(x86)%\Platform-MPI
```

Note:

This is the default location on 64-bit machines. The location for 32-bit machines is `%ProgramFiles%\Platform-MPI`

The MPI application can now be built with Platform MPI.

The property page sets the following fields automatically, but they can be set manually if the property page provided is not used:

- C/C++ ??: Additional Include Directories

  Set to `"\%MPI_ROOT%\include\[32|64]"`
- Linker : Additional Dependencies

  Set to `libpcmpi32.lib` or `libpcmpi64.lib` depending on the application.
- Additional Library Directories

  Set to `"%MPI_ROOT%\lib"`

Building and running on a Windows 2008 cluster using appfiles

The example teaches you the basic compilation and run steps to execute `hello_world.c` on a cluster with 4-way parallelism.

Note:

Specify the bitness using -mpi64 or -mpi32 for `mpicc` to link in the correct libraries. Verify you are in the correct bitness compiler window. Using -mpi64 in a Visual Studio 32-bit command window does not work.

1. Create a file "appfile" for running on nodes n01 and n02 as:

   -h n01 -np 2 \\node01\share\path\to\hello_world.exe

   -h n02 -np 2 \\node01\share\path\to\hello_world.exe
2. For the first run of the hello_world executable, use -cache to cache your password:

   "%MPI_ROOT%\bin\mpirun" -cache -f appfile

   ```
   Password for MPI runs:
   ```

   When typing, the password is not echoed to the screen.

   The Platform MPI Remote Launch service must be registered and started on the remote nodes. `mpirun` will authenticated with the service and create processes using your encrypted password to obtain network resources.

   If you do not provide a password, the password is incorrect, or you use -nopass, remote processes are created but do not have access to network shares. In the following example, the `hello_world.exe` file cannot be read.
3. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

   ```
   Hello world! I'm 1 of 4 on n01
   Hello world! I'm 3 of 4 on n02
   Hello world! I'm 0 of 4 on n01
   Hello world! I'm 2 of 4 on n02
   ```

Running with an appfile using HPCS

Using an appfile with HPCS has been greatly simplified in this release of Platform MPI. The previous method of writing a submission script that uses `mpi_nodes.exe` to dynamically generate an appfile based on the HPCS allocation is still supported. However, the preferred method is to allow `mpirun.exe` to determine which nodes are required for the job (by reading the user-supplied appfile), request those nodes from the HPCS scheduler, then submit the job to HPCS when the requested nodes have been allocated. Users write a brief appfile calling out the exact nodes and rank counts needed for the job. For example:

1. Change to a writable directory.
2. Compile the `hello_world` executable file:

   `%` **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**
3. Create an appfile for running on nodes n01 and n02 as:

   -h n01 -np 2 hello_world.exe

   -h n02 -np 2 hello_world.exe
4. Submit the job to HPCS with the following command:

   `X:\demo>` **mpirun -hpc -f appfile**
5. Analyze `hello_world` output.

   Platform MPI prints the output from running the `hello_world` executable in non-deterministic order. The following is an example of the output.

```
Hello world! I'm 2 of 4 on n02
Hello world! I'm 1 of 4 on n01
Hello world! I'm 0 of 4 on n01
Hello world! I'm 3 of 4 on n02
```

More information about using appfiles is available in Chapter 3 of the *Platform MPI User's Guide*.

Building and running on a Windows 2003/XP cluster using appfiles

The following example shows the basic compilation and run steps to execute hello_world.c on a cluster with 4-way parallelism. To build and run `hello_world.c` on a cluster using an appfile:

Note:

Specify the bitness using -mpi64 or -mpi32 for `mpicc` to link in the correct libraries. Verify you are in the correct bitness compiler window. Using -mpi64 in a Visual Studio 32-bit command window will not work.

1. Change to a writable directory.
2. Compile the `hello_world` executable file:

   `>` **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**
3. Create the file `appfile` for running on nodes n01 and n02 as:

   -h n01 -np 2 \\node01\share\path\to\hello_world.exe

   -h n02 -np 2 \\node01\share\path\to\hello_world.exe
4. For the first run of the
   `hello_world`

executable, use -cache to cache your password:

> **"%MPI_ROOT%\bin\mpirun" -cache -f appfile**

```
Password for MPI runs:
```

When typing, the password is not echoed to the screen.

The Platform MPI Remote Launch service must be registered and started on the remote nodes. `mpirun` authenticates with the service and creates processes using your encrypted password to obtain network resources.

If you do not provide a password, if the password is incorrect, or if you use -nopass, remote processes are created but do not have access to network shares. (In this example, the remote process cannot read the `hello_world.exe` file.)

5. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in nondeterministic order. The following is an example of the output:

```
Hello world! I'm 1 of 4 on n01
Hello world! I'm 3 of 4 on n02
Hello world! I'm 0 of 4 on n01
Hello world! I'm 2 of 4 on n02
```

Directory structure for Windows

All Platform MPI for Windows files are stored in the directory specified at installation. The default directory is `C:\Program Files (x86)\Platform-MPI`. If you move the Platform MPI installation directory from its default location, set the *MPI_ROOT* environment variable to point to the new location. The directory structure is organized as follows:

Table 8: Directory structure for Windows

Subdirectory	Contents
bin	Command files for Platform MPI utilities
help	Source files for example programs and Visual Studio Property pages
include\32	32-bit header files
include\64	64-bit header files
lib	Platform MPI libraries
man	Platform MPI manpages in HTML format
devtools	Windows Platform MPI services
licenses	Repository for Platform MPI license file
doc	Release notes and the Debugging with Platform MPI Tutorial

Windows man pages

The manpages are located in the `"%MPI_ROOT%\man\"` subdirectory for Windows. They can be grouped into three categories: general, compilation, and run-time. One general manpage, MPI.1, is an overview

describing general features of Platform MPI. The compilation and run-time manpages describe Platform MPI utilities.

The following table describes the three categories of manpages in the man1 subdirectory that comprise manpages for Platform MPI utilities:

Table 9: Windows man page categories

Category	manpages	Description
General	MPI.1	Describes the general features of Platform MPI.
Compilation	• mpicc.1 • mpif90.1	Describes the available compilation utilities.
Run time	• mpidebug.1 • mpienv.1 • mpimtsafe.1 • mpirun.1 • mpistdio.1 • autodbl.1	Describes run-time utilities, environment variables, debugging, thread-safe, and diagnostic libraries.

Licensing policy for Windows

Platform MPI for Windows uses FlexNet Publishing (formerly FLEXlm) licensing technology. A license is required to use Platform MPI for Windows. Licenses can be can be acquired from Platform Computing.

Platform MPI has an Independent Software Vendor (ISV) program that allows participating ISVs to freely distribute Platform MPI with their applications. When the application is part of the Platform MPI ISV program, there is no licensing requirement for the user. The ISV provides a licensed copy of Platform MPI. Contact your application vendor to find out if they participate in the Platform MPI ISV program. The copy of Platform MPI distributed with a participating ISV only works with that application. A Platform MPI license is required for all other applications.

Licensing for Windows

Platform MPI for Windows uses FlexNet Publisher (formerly FLEXlm) licensing technology. A license file can be named `license.dat` or any file name with an extension of `.lic`. The license file must be placed in the installation directory (default `C:\Program Files (x86)\Platform-MPI\licenses`) on all run-time systems, and on the license server.

Platform MPI for Windows optionally supports redundant license servers. The Platform MPI License Certificate includes space for up to three license servers. Either one license server, or three license servers are listed on the certificate. To use a single license server, follow the directions below. To use three redundant license servers, repeat the steps below for each license server.

You must provide the host name and host ID number of the system where the FlexNet daemon for Platform MPI for Windows will run. The host ID can be obtained by entering the following command if Platform MPI is already installed on the system:

%MPI_ROOT%\bin\licensing*i86_n3*\lmutil lmhostid

To obtain the host name, use the control panel by using Control Panel > System > Computer Name.

The default search path used to find an MPI license file is:

"%MPI_ROOT%\licenses:.".

If the license must be placed in a location that would not be found by the above search, the you can set the environment variable *LM_LICENSE_FILE* to specify the location of the license file.

Installing license files

A valid license file contains the system host ID and the associated license key. License files can be named `license.dat` or any name with extension of *.`lic` (For example, `mpi.lic`). The license file must be copied to the installation directory (default `C:\Program Files (x86)\Platform-MPI\licenses`) on all run-time systems, and to the license server.

The command to run the license server is:

```
% "MPI_ROOT%\bin\licensing\<i86_n3>\lmgrd" -c mpi.lic
```

License testing

To check for a license, build and run the hello_world program in `%MPI_ROOT%\help\hello_world.c`. If your system is not properly licensed, you will receive the following error message:

```
(MPI BUG: Valid MPI license not found in search path)
```

CHAPTER 4

Understanding Platform MPI

This chapter provides information about the Platform MPI implementation of MPI.

Compilation wrapper script utilities

Platform MPI provides compilation utilities for the languages shown in the following table. In general, if a specific compiler is needed, set the related environment variable, such as *MPI_CC*. Without such a setting, the utility script searches the *PATH* and a few default locations for possible compilers. Although in many environments this search produces the desired results, explicitly setting the environment variable is safer. Command-line options take precedence over environment variables.

Table 10: Compiler selection

Language	Wrapper Script	Environment Variable	Command Line
C	`mpicc`	*MPI_CC*	-mpicc <*compiler*>
C++	`mpiCC`	*MPI_CXX*	-mpicxx <*compiler*>
Fortran 77	`mpif77`	*MPI_F77*	-mpif77 <*compiler*>
Fortran 90	`mpif90`	*MPI_F90*	-mpif90 <*compiler*>

Compiling applications

The compiler you use to build Platform MPI applications depends on the programming language you use. Platform MPI compiler utilities are shell scripts that invoke the correct native compiler. You can pass the pathname of the MPI header files using the -I option and link an MPI library (for example, the diagnostic or thread-compliant library) using the -Wl, -L or -l option.

By default, Platform MPI compiler utilities include a small amount of debug information to allow the TotalView debugger to function. However, some compiler options are incompatible with this debug information. Use the -notv option to exclude debug information. The -notv option also disables TotalView usage on the resulting executable. The -notv option applies to archive libraries only.

Platform MPI offers a -show option to compiler wrappers. When compiling by hand, run `mpicc -show` and a line prints showing what the job would do (and skipping the build).

Fortran 90

To use the 'mpi' Fortran 90 module, you must create the module file by compiling the `module.F` file in `/opt/platform_mpi/include/64/module.F` for 64-bit compilers. For 32-bit compilers, compile the `module.F` file in `/opt/platform_mpi/include/32/module.F`.

Note:

Each vendor (e.g., PGI, Qlogic/Pathscale, Intel, Gfortran, etc.) has a different module file format. Because compiler implementations vary in their representation of a module file, a PGI module file is not usable with Intel and so on. Additionally, forward compatibility might not be the case from older to newer versions of a specific vendor's compiler. Because of compiler version compatibility and format issues, we do not build module files.

In each case, you must build (just once) the module that corresponds to 'mpi' with the compiler you intend to use.

For example, with `platform_mpi/bin` and `pgi/bin` in `path`:

```
pgf90 -c  /opt/platform_mpi/include/64/module.F
cat >hello_f90.f90 program main
     use mpi
       implicit none
       integer :: ierr, rank, size
     call MPI_INIT(ierr)
       call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
       call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
       print *, "Hello, world, I am ", rank, " of ", size
       call MPI_FINALIZE(ierr)
End
mpif90 -mpif90 pgf90 hello_f90.f90
hello_f90.f90:
mpirun ./a.out
Hello, world, I am 0 of 1
```

C command-line basics for Windows

The utility `"%MPI_ROOT%\bin\mpicc"` is included to aid in command-line compilation. To compile with this utility, set the *MPI_CC* environment variable to the path of the command-line compiler you want to use. Specify -mpi32 or -mpi64 to indicate if you are compiling a 32-bit or 64-bit application. Specify the command-line options that you would normally pass to the compiler on the `mpicc` command line. The `mpicc` utility adds command-line options for Platform MPI include directories and libraries. You can specify the -show option to indicate that `mpicc` should display the command generated without executing the compilation command. For more information, see the `mpicc` manpage .

To construct the compilation command, the `mpicc` utility must know what command-line compiler is to be used, the bitness of the executable that compiler will produce, and the syntax accepted by the compiler. These can be controlled by environment variables or from the command line.

Table 11: `mpicc` **utility**

Environment Variable	Value	Command Line
MPI_CC	desired compiler (default cl)	-mpicc <*value*>
MPI_BITNESS	32 or 64 (no default)	-mpi32 or -mpi64
MPI_WRAPPER_SYNTAX	windows or unix (default windows)	-mpisyntax <*value*>

For example, to compile hello_world.c with a 64-bit 'cl' contained in your *PATH* use the following command because 'cl' and the 'Windows' syntax are defaults:

"%MPI_ROOT%\bin\mpicc" /mpi64 hello_world.c /link /out:hello_world_cl64.exe

Or, use the following example to compile using the PGI compiler, which uses a more UNIX-like syntax:

"%MPI_ROOT%\bin\mpicc" -mpicc pgcc -mpisyntax unix -mpi32 hello_world.c -o hello_world_pgi32.exe

To compile C code and link with Platform MPI without using the `mpicc` tool, start a command prompt that has the relevant environment settings loaded for your compiler, and use it with the compiler option:

/I"%MPI_ROOT%\include\[32|64]"

and the linker options:

/libpath:"%MPI_ROOT%\lib" /subsystem:console [libpcmpi64.lib|libpcmpi32.lib]

Specify bitness where indicated. The above assumes the environment variable *MPI_ROOT* is set.

For example, to compile hello_world.c from the `%MPI_ROOT%\help` directory using Visual Studio (from a Visual Studio 2005 command prompt window):

cl hello_world.c /I"%MPI_ROOT%\include\64" /link /out:hello_world.exe /libpath:"%MPI_ROOT%\lib" /subsystem:console libpcmpi64.lib

The PGI compiler uses a more UNIX-like syntax. From a PGI command prompt:

pgcc hello_world.c -I"%MPI_ROOT%\include\64" -o hello_world.exe -L"%MPI_ROOT%\lib" -lhpmpi64

Fortran command-line basics for Windows

The utility `"%MPI_ROOT%\bin\mpif90"` is included to aid in command-line compilation. To compile with this utility, set the *MPI_F90* environment variable to the path of the command-line compiler you want to use. Specify -mpi32 or -mpi64 to indicate if you are compiling a 32-bit or 64-bit application. Specify the command-line options that you would normally pass to the compiler on the `mpif90` command line. The `mpif90` utility adds additional command-line options for Platform MPI include directories and libraries. You can specify the -show option to indicate that `mpif90` should display the command generated without executing the compilation command. For more information, see the `mpif90` manpage.

To construct the compilation command, the `mpif90` utility must know what command-line compiler is to be used, the bitness of the executable that compiler will produce, and the syntax accepted by the compiler. These can be controlled by environment variables or from the command line.

Table 12: `mpif90` **utility**

Environment Variable	Value	Command Line
MPI_F90	desired compiler (default ifort)	-mpif90 <*value*>
MPI_BITNESS	32 or 64 (no default)	-mpi32 or -mpi64
MPI_WRAPPER_SYNTAX	windows or unix (default windows)	-mpisyntax <*value*>

For example, to compile `compute_pi.f` with a 64-bit ifort contained in your *PATH* use the following command because ifort and the Windows syntax are defaults:

"%MPI_ROOT%\bin\mpif90" /mpi64 compute_pi.f /link /out:compute_pi_ifort.exe

Or, use the following example to compile using the PGI compiler, which uses a more UNIX-like syntax:

"%MPI_ROOT%\bin\mpif90" -mpif90 pgf90 -mpisyntax unix -mpi32 compute_pi.f -o compute_pi_pgi32.exe

To compile `compute_pi.f` using Intel Fortran without using the `mpif90` tool (from a command prompt that has the relevant environment settings loaded for your Fortran compiler):

ifort compute_pi.f /I"%MPI_ROOT%\include\64" /link /out:compute_pi.exe /libpath:"%MPI_ROOT%\lib" /subsystem:console libpcmpi64.lib

Note:

Intel compilers often link against the Intel runtime libraries. When running an MPI application built with the Intel Fortran or C/C++ compilers, you might need to install the Intel run-time libraries on every node of your cluster. We recommend that you install the version of the Intel run-time

libraries that correspond to the version of the compiler used on the MPI application.

C++ bindings (for Linux)

Platform MPI supports C++ bindings as described in the MPI-2 Standard. If you compile and link with the `mpiCC` command, no additional work is needed to include and use the bindings. You can include `mpi.h` or `mpiCC.h` in your C++ source files.

The bindings provided by Platform MPI are an interface class, calling the equivalent C bindings. To profile your application, you should profile the equivalent C bindings.

If you build without the `mpiCC` command, include -lmpiCC to resolve C++ references.

To use an alternate `libmpiCC.a` with `mpiCC`, use the -mpiCClib <*LIBRARY*> option. A 'default' g++ ABI-compatible library is provided for each architecture except Alpha.

Non-g++ ABI compatible C++ compilers

The C++ library provided by Platform MPI, `libmpiCC.a`, was built with g++. If you are using a C++ compiler that is not g++ ABI compatible (e.g., Portland Group Compiler), you must build your own `libmpiCC.a` and include this in your build command. The sources and Makefiles to build an appropriate library are located in `/opt/platform_mpi/lib/ARCH/mpiCCsrc`.

To build a version of `libmpiCC.a` and include it in the builds using `mpiCC`, do the following:

Note:

This example assumes your Platform MPI installation directory is `/opt/platform_mpi`. It also assumes that the pgCC compiler is in your path and working properly.

1. Copy the file needed to build `libmpiCC.a` into a working location.

   % **setenv MPI_ROOT /opt/platform_mpi**

   % **cp -r $MPI_ROOT/lib/linux_amd64/mpiCCsrc ~**

   % **cd ~/mpiCCsrc**
2. Compile and create the `libmpiCC.a` library.

   % **make CXX=pgCC MPI_ROOT=$MPI_ROOT**

   ```
   pgCC -c intercepts.cc -I/opt/platform_mpi/include -DHPMP_BUILD_CXXBINDING
   PGCC-W-0155-Nova_start() seen (intercepts.cc:33) PGCC/x86 Linux/x86-64
   6.2-3: compilation completed with warnings pgCC -c mpicxx.cc - I/opt/
   platform_mpi/include -DHPMP_BUILD_CXXBINDING ar rcs libmpiCC.a
   intercepts.o mpicxx.o
   ```
3. Using a test case, verify that the library works as expected.

   % **mkdir test ; cd test**

   % **cp $MPI_ROOT/help/sort.C .**

   % **$MPI_ROOT/bin/mpiCC HPMPI_CC=pgCC sort.C -mpiCClib \ ../libmpiCC.a**

   ```
   sort.C:
   ```

   % **$MPI_ROOT/bin/mpirun -np 2 ./a.out**

   ```
   Rank 0
   -980
   -980
   ```

```
        .
        .
        .
965
965
```

Autodouble functionality

Platform MPI supports Fortran programs compiled 64-bit with any of the following options (some of which are not supported on all Fortran compilers):

For Linux:

- -i8

  Set default KIND of integer variables is 8.
- -r8

  Set default size of REAL to 8 bytes.
- -r16

  Set default size of REAL to 16 bytes.
- -autodouble

  Same as -r8.

The decision of how Fortran arguments are interpreted by the MPI library is made at link time.

If the `mpif90` compiler wrapper is supplied with one of the above options at link time, the necessary object files automatically link, informing MPI how to interpret the Fortran arguments.

Note:

This autodouble feature is supported in the regular and multithreaded MPI libraries, but not in the diagnostic library.

For Windows:

- /integer_size:64
- /4I8
- -i8
- /real_size:64
- /4R8
- /Qautodouble
- -r8

If these flags are given to the `mpif90.bat` script at link time, the application is linked, enabling Platform MPI to interpret the data type *MPI_REAL* as 8 bytes (etc. as appropriate) at run time.

However, if your application is written to explicitly handle autodoubled datatypes (e.g., if a variable is declared real, the code is compiled -r8, and corresponding MPI calls are given *MPI_DOUBLE* for the datatype), then the autodouble related command-line arguments should not be passed to `mpif90.bat` at link time (because that causes the datatypes to be automatically changed).

MPI functions

The following MPI functions accept user-defined functions and require special treatment when autodouble is used:

- MPI_Op_create()
- MPI_Errhandler_create()
- MPI_Keyval_create()
- MPI_Comm_create_errhandler()
- MPI_Comm_create_keyval()
- MPI_Win_create_errhandler()
- MPI_Win_create_keyval()

The user-defined callback passed to these functions should accept normal-sized arguments. These functions are called internally by the library where normally-sized data types are passed to them.

64-bit support

Platform MPI provides support for 64-bit libraries as shown below. More information about Linux and Windows systems is provided in the following sections.

Table 13: 32-bit and 64-bit support

OS/Architecture	Supported Libraries	Default	Notes
Linux IA-32	32-bit	32-bit	
Linux Itanium2	64-bit	64-bit	
Linux Opteron & Intel64	32-bit and 64-bit	64-bit	Use -mpi32 and appropriate compiler flag. For 32-bit flag, see the compiler manpage.
Windows	32-bit and 64-bit	N/A	

Linux

Platform MPI supports 32-bit and 64-bit versions running Linux on AMD Opteron or Intel64 systems. 32-bit and 64-bit versions of the library are shipped with these systems; however, you cannot mix 32-bit and 64-bit executables in the same application.

Platform MPI includes -mpi32 and -mpi64 options for the compiler wrapper script on Opteron and Intel64 systems. Use these options to indicate the bitness of the application to be invoked so that the availability of interconnect libraries can be properly determined by the Platform MPI utilities `mpirun` and `mpid`. The default is -mpi64.

Windows

Platform MPI supports 32-bit and 64-bit versions running Windows on AMD Opteron or Intel64. 32-bit and 64-bit versions of the library are shipped with these systems; however you cannot mix 32-bit and 64-bit executables in the same application.

Platform MPI includes -mpi32 and -mpi64 options for the compiler wrapper script on Opteron and Intel64 systems. These options are only necessary for the wrapper scripts so the correct `libpcmpi32.dll` or `libpcmpi64.dll` file is linked with the application. It is not necessary when invoking the application.

Thread-compliant library

Platform MPI provides a thread-compliant library. By default, the non thread-compliant library (libmpi) is used when running Platform MPI jobs. Linking to the thread-compliant library is required only for applications that have multiple threads making MPI calls simultaneously. In previous releases, linking to the thread-compliant library was required for multithreaded applications even if only one thread was making a MPI call at a time.

To link with the thread-compliant library on Linux systems, specify the -libmtmpi option to the build scripts when compiling the application.

To link with the thread-compliant library on Windows systems, specify the -lmtmpi option to the build scripts when compiling the application.

Application types that no longer require linking to the thread-compliant library include:

- Implicit compiler-generated parallelism.
- Thread parallel applications using the HP MLIB math libraries.
- OpenMP applications.
- pthreads (Only if no two threads call MPI at the same time. Otherwise, use the thread-compliant library for pthreads.)

CPU binding

The `mpirun` option -cpu_bind binds a rank to an ldom to prevent a process from moving to a different ldom after start-up. The binding occurs before the MPI application is executed.

To accomplish this, a shared library is loaded at start-up that does the following for each rank:

- Spins for a short time in a tight loop to let the operating system distribute processes to CPUs evenly. This duration can be changed by setting the *MPI_CPU_SPIN* environment variable which controls the number of spins in the initial loop. Default is 3 seconds.
- Determines the current CPU and ldom.
- Checks with other ranks in the MPI job on the host for oversubscription by using a "shm" segment created by `mpirun` and a lock to communicate with other ranks. If no oversubscription occurs on the current CPU, then lock the process to the ldom of that CPU. If a rank is reserved on the current CPU, find a new CPU based on least loaded free CPUs and lock the process to the ldom of that CPU.

Similar results can be accomplished using "mpsched" but the procedure outlined above is a more load-based distribution and works well in psets and across multiple machines.

Platform MPI supports CPU binding with a variety of binding strategies (see below). The option -cpu_bind is supported in appfile, command-line, and `srun` modes.

```
% mpirun -cpu_bind[_mt]=[v,][option][,v] -np \ 4 a.out
```

Where _mt implies thread aware CPU binding; v, and ,v request verbose information on threads binding to CPUs; and [*option*] is one of:

rank : Schedule ranks on CPUs according to packed rank ID.

map_cpu : Schedule ranks on CPUs in cyclic distribution through *MAP* variable.

mask_cpu : Schedule ranks on CPU masks in cyclic distribution through *MAP* variable.

ll : least loaded (ll) Bind each rank to the CPU it is running on.

For NUMA-based systems, the following options are also available:

ldom : Schedule ranks on ldoms according to packed rank ID.

cyclic : Cyclic dist on each ldom according to packed rank ID.

block : Block dist on each ldom according to packed rank ID.

rr : round robin (rr) Same as cyclic, but consider ldom load average.

fill : Same as block, but consider ldom load average.

packed : Bind all ranks to same ldom as lowest rank.

slurm : slurm binding.

ll : least loaded (ll) Bind each rank to ldoms it is running on.

map_ldom : Schedule ranks on ldoms in cyclic distribution through MAP variable.

To generate the current supported options:

```
% mpirun -cpu_bind=help ./a.out
```

Environment variables for CPU binding:

- *MPI_BIND_MAP* allows specification of the integer CPU numbers, ldom numbers, or CPU masks. These are a list of integers separated by commas (,).
- *MPI_CPU_AFFINITY* is an alternative method to using -cpu_bind on the command line for specifying binding strategy. The possible settings are LL, RANK, MAP_CPU, MASK_CPU, LDOM, CYCLIC, BLOCK, RR, FILL, PACKED, SLURM, AND MAP_LDOM.
- *MPI_CPU_SPIN* allows selection of spin value. The default is 2 seconds. This value is used to let busy processes spin so that the operating system schedules processes to processors. The processes bind themselves to the relevant processor, or core, or ldom.

  For example, the following selects a 4-second spin period to allow 32 MPI ranks (processes) to settle into place and then bind to the appropriate processor/core/ldom.

  % **mpirun -e MPI_CPU_SPIN=4 -cpu_bind -np\ 32 ./linpack**
- *MPI_FLUSH_FCACHE* can be set to a threshold percent of memory (0-100) which, if the file cache currently in use meets or exceeds, initiates a flush attempt after binding and essentially before the user's MPI program starts.
- *MPI_THREAD_AFFINITY* controls thread affinity. Possible values are:

  none : Schedule threads to run on all cores/ldoms. This is the default.

  cyclic : Schedule threads on ldoms in cyclic manner starting after parent.

  cyclic_cpu : Schedule threads on cores in cyclic manner starting after parent.

  block : Schedule threads on ldoms in block manner starting after parent.

  packed : Schedule threads on same ldom as parent.

  empty : No changes to thread affinity are made.
- *MPI_THREAD_IGNSELF* when set to yes, does not include the parent in scheduling consideration of threads across remaining cores/ldoms. This method of thread control can be used for explicit pthreads or OpenMP threads.

Three -cpu_bind options require the specification of a map/mask description. This allows for explicit binding of ranks to processors. The three options are map_ldom, map_cpu, and mask_cpu.

Syntax:

-cpu_bind=[map_ldom,map_cpu,mask_cpu] [:<*settings*>, =<*settings*>, -e MPI_BIND_MAP=<*settings*>]

Examples:

-cpu_bind=MAP_LDOM -e MPI_BIND_MAP=0,2,1,3

map rank 0 to ldom 0, rank 1 to ldom 2, rank 2 to ldom1 and rank 3 to ldom 3.

-cpu_bind=MAP_LDOM=0,2,3,1

map rank 0 to ldom 0, rank 1 to ldom 2, rank 2 to ldom 3 and rank 3 to ldom 1.

-cpu_bind=MAP_CPU:0,6,5

map rank 0 to cpu 0, rank 1 to cpu 6, rank 2 to cpu 5.

-cpu_bind=MASK_CPU:1,4,6

map rank 0 to cpu 0 (0001), rank 1 to cpu 2 (0100), rank 2 to cpu 1 or 2 (0110).

A rank binding on a clustered system uses the number of ranks and the number of nodes combined with the rank count to determine CPU binding. Cyclic or blocked launch is taken into account.

On a cell-based system with multiple users, the LL strategy is recommended rather than RANK. LL allows the operating system to schedule computational ranks. Then the -cpu_bind capability locks the ranks to the CPU as selected by the operating system scheduler.

MPICH object compatibility for Linux

The MPI standard specifies the function prototypes for MPI functions but does not specify types of MPI opaque objects like communicators or the values of MPI constants. As a result, an object file compiled using one vendor's MPI generally does not function if linked to another vendor's MPI library.

There are some cases where such compatibility would be desirable. For instance a third-party tool such as Intel trace/collector might only be available using the MPICH interface.

To allow such compatibility, Platform MPI includes a layer of MPICH wrappers. This provides an interface identical to MPICH 1.2.5, and translates these calls into the corresponding Platform MPI interface. This MPICH compatibility interface is only provided for functions defined in MPICH 1.2.5 and cannot be used by an application that calls functions outside the scope of MPICH 1.2.5.

Platform MPI can be used in MPICH mode by compiling using `mpicc.mpich` and running using `mpirun.mpich`. The compiler script `mpicc.mpich` uses an include file that defines the interfaces the same as MPICH 1.2.5, and at link time it links against `libmpich.so` which is the set of wrappers defining MPICH 1.2.5 compatible entry points for the MPI functions. The `mpirun.mpich` takes the same arguments as the traditional Platform MPI `mpirun`, but sets *LD_LIBRARY_PATH* so that `libmpich.so` is found.

An example of using a program with Intel Trace Collector:

```
% export MPI_ROOT=/opt/platform_mpi
```

```
% $MPI_ROOT/bin/mpicc.mpich -o prog.x $MPI_ROOT/help/communicator.c -L/path/to/itc/lib -IVT
-lvtunwind -ldwarf -lnsl -lm -lelf -lpthread
```

```
% $MPI_ROOT/bin/mpirun.mpich -np 2 ./prog.x
```

Here, the program communicator.c is compiled with MPICH compatible interfaces and is linked to Intel's Trace Collector `libVT.a` first from the command-line option, followed by Platform MPI's `libmpich.so` and then `libmpi.so` which are added by the `mpicc.mpich` compiler wrapper script. Thus `libVT.a` sees only the MPICH compatible interface to Platform MPI.

In general, object files built with Platform MPI's MPICH mode can be used in an MPICH application, and conversely object files built under MPICH can be linked into a Platform MPI application using MPICH mode. However, using MPICH compatibility mode to produce a single executable to run under MPICH and Platform MPI can be problematic and is not advised.

You can compile communicator.c under Platform MPI MPICH compatibility mode as:

```
% export MPI_ROOT=/opt/platform_mpi
```

```
% $MPI_ROOT/bin/mpicc.mpich -o prog.x\$MPI_ROOT/help/communicator.c
```

and run the resulting prog.x under MPICH. However, some problems will occur. First, the MPICH installation must be built to include shared libraries and a soft link must be created for `libmpich.so`, because their libraries might be named differently.

Next an appropriate *LD_LIBRARY_PATH* setting must be added manually because MPICH expects the library path to be hard-coded into the executable at link time via -rpath.

Finally, although the resulting executable can run over any supported interconnect under Platform MPI, it will not under MPICH due to not being linked to libgm/libelan etc.

Similar problems would be encountered if linking under MPICH and running under Platform MPI's MPICH compatibility. MPICH's use of -rpath to hard-code the library path at link time keeps the

executable from being able to find the Platform MPI MPICH compatibility library via Platform MPI's *LD_LIBRARY_PATH* setting.

C++ bindings are not supported with MPICH compatibility mode.

MPICH compatibility mode is not supported on Platform MPI V1.0 for Windows.

MPICH2 compatibility

MPICH compatibility mode supports applications and libraries that use the MPICH2 implementation. MPICH2 is not a standard, but rather a specific implementation of the MPI-2.1 standard. Platform MPI provides MPICH2 compatibility with the following wrappers:

Table 14: MPICH wrappers

MPICH1	MPICH2
`mpirun.mpich`	`mpirun.mpich2`
`mpicc.mpich`	`mpicc.mpich2`
`mpif77.mpich`	`mpif77.mpich2`
`mpif90.mpich`	`mpif90.mpich2`

Object files built with Platform MPI MPICH compiler wrappers can be used by an application that uses the MPICH implementation. You must relink applications built using MPICH compliant libraries to use Platform MPI in MPICH compatibility mode.

Note:

Do not use MPICH compatibility mode to produce a single executable to run under both MPICH and Platform MPI.

Examples of building on Linux

This example shows how to build hello_world.c prior to running.

1. Change to a writable directory that is visible from all hosts the job will run on.
2. Compile the hello_world executable file.

   % **$MPI_ROOT/bin/mpicc -o hello_world $MPI_ROOT/help/hello_world.c**

   This example uses shared libraries, which is recommended.

   Platform MPI also includes archive libraries that can be used by specifying the correct compiler option.

   Note:

   Platform MPI uses the dynamic loader to interface with interconnect libraries. Therefore, dynamic linking is required when building applications that use Platform MPI.

Running applications on Linux

This section introduces the methods to run your Platform MPI application on Linux. Using an `mpirun` method is required. The examples below demonstrate six basic methods. For all the `mpirun` command-line options, refer to the `mpirun` documentation.

Platform MPI includes -mpi32 and -mpi64 options for the launch utility `mpirun` on Opteron and Intel64. Use these options to indicate the bitness of the application to be invoked so that the availability of interconnect libraries can be correctly determined by the Platform MPI utilities `mpirun` and `mpid`. The default is -mpi64.

You can use one of six methods to start your application, depending on what the system you are using:

- Use `mpirun` with the -np# option and the name of your program. For example,

  % **$MPI_ROOT/bin/mpirun -np 4 hello_world**

  starts an executable file named `hello_world` with four processes. This is the recommended method to run applications on a single host with a single executable file.
- Use `mpirun` with an appfile. For example:

  % **$MPI_ROOT/bin/mpirun -f appfile**

  where -f appfile specifies a text file (appfile) that is parsed by `mpirun` and contains process counts and a list of programs. Although you can use an appfile when you run a single executable file on a single host, it is best used when a job is to be run across a cluster of machines that does not have a dedicated launching method such as `srun` or `prun` (described below), or when using multiple executables.
- Use `mpirun` with -prun using the Quadrics Elan communication processor on Linux. For example:

  % **$MPI_ROOT/bin/mpirun [*mpirun options*] -prun <*prun options*> <*program*> <*args*>**

  This method is only supported when linking with shared libraries.

  Some features like `mpirun -stdio` processing are unavailable.

  Rank assignments within Platform MPI are determined by the way `prun` chooses mapping at run time.

  The -np option is not allowed with -prun. The following `mpirun` options are allowed with -prun:

  % **$MPI_ROOT/bin/mpirun [-help] [-version] [-jv] [-i <spec>] [-universe_size=#] [-sp <*paths*>] [-T] [-prot] [-spawn] [-1sided] [-tv] [-e var[=val]] -prun <*prun options*> <*program*> [<*args*>]**

  For more information on `prun` usage:

  % **man prun**

  The following examples assume the system has the Quadrics Elan interconnect and is a collection of 2-CPU nodes.

  % **$MPI_ROOT/bin/mpirun -prun -N4 ./a.out**

  will run `a.out` with 4 ranks, one per node. Ranks are cyclically allocated.

```
n00 rank1
n01 rank2
n02 rank3
n03 rank4
```

  % **$MPI_ROOT/bin/mpirun -prun -n4 ./a.out**

(assuming nodes have 2 processors/cores each) will run `a.out` with 4 ranks, 2 ranks per node, ranks are block allocated. Two nodes are used.

```
n00 rank1
n01 rank2
n02 rank3
n03 rank4
```

Other forms of usage include allocating the nodes you want to use, which creates a subshell. Then jobsteps can be launched within that subshell until the subshell is exited.

```
% $MPI_ROOT/bin/mpirun -prun -A -N6
```

This allocates 6 nodes and creates a subshell.

```
% $MPI_ROOT/bin/mpirun -prun -n4 -m block ./a.out
```

This uses 4 ranks on 4 nodes from the existing allocation. Note that we asked for block.

```
n00 rank1
n01 rank2
n02 rank3
n03 rank4
```

- Use `mpirun` with -srun on HP XC LSF clusters. For example:

  % $MPI_ROOT/bin/mpirun <*mpirun options*> -srun <*srun options*> <*program*> <*args*>

  Some features like `mpirun -stdio` processing are unavailable.

  The -np option is not allowed with -srun. The following options are allowed with -srun:

  % $MPI_ROOT/bin/mpirun [-help] [-version] [-jv] [-i <spec>] [-universe_size=#] [-sp <*paths*>] [-T] [-prot] [-spawn] [-tv] [-1sided] [-e var[=val]] -srun <*srun options*> <*program*> [<*args*>]

  For more information on `srun` usage:

  % man srun

  The following examples assume the system has the Quadrics Elan interconnect, SLURM is configured to use Elan, and the system is a collection of 2-CPU nodes.

  % $MPI_ROOT/bin/mpirun -srun -N4 ./a.out

  ill run `a.out`with 4 ranks, one per node. Ranks are cyclically allocated.

  ```
  n00 rank1
  n01 rank2
  n02 rank3
  n03 rank4
  ```

  % $MPI_ROOT/bin/mpirun -srun -n4 ./a.out

  will run `a.out` with 4 ranks, 2 ranks per node, ranks are block allocated. Two are nodes used.

  Other forms of usage include allocating the nodes you want to use, which creates a subshell. Then jobsteps can be launched within that subshell until the subshell is exited.

  % srun -A -n4

  This allocates 2 nodes with 2 ranks each and creates a subshell.

  % $MPI_ROOT/bin/mpirun -srun ./a.out

  This runs on the previously allocated 2 nodes cyclically.

  ```
  n00 rank1
  n01 rank2
  n02 rank3
  n03 rank4
  ```

- Use HP XC LSF and Platform MPI

  Platform MPI jobs can be submitted using LSF. LSF uses the SLURM `srun` launching mechanism. Because of this, Platform MPI jobs must specify the -srun option whether LSF is used or `srun` is used.

  % bsub -I -n2 $MPI_ROOT/bin/mpirun -srun ./a.out

  LSF creates an allocation of 2 processors and `srun` attaches to it.

  % bsub -I -n12 $MPI_ROOT/bin/mpirun -srun -n6 -N6 ./a.out

  LSF creates an allocation of 12 processors and `srun` uses 1 CPU per node (6 nodes). Here, we assume 2 CPUs per node.

  LSF jobs can be submitted without the -I (interactive) option.

  An alternative mechanism for achieving the one rank per node which uses the -ext option to LSF:

  % bsub -I -n3 -ext "SLURM[nodes=3]" $MPI_ROOT/bin/mpirun -srun ./a.out

  The -ext option can also be used to specifically request a node. The command line would look something like the following:

  % bsub -I -n2 -ext "SLURM[nodelist=n10]" mpirun -srun ./hello_world

```
Job <1883> is submitted to default queue <interactive>.
<<Waiting for dispatch ...>>
<<Starting on lsfhost.localdomain>>
Hello world! I'm 0 of 2 on n10
Hello world! I'm 1 of 2 on n10
```

  Including and excluding specific nodes can be accomplished by passing arguments to SLURM as well. For example, to make sure a job includes a specific node and excludes others, use something like the following. In this case, n9 is a required node and n10 is specifically excluded:

  % bsub -I -n8 -ext "SLURM[nodelist=n9;exclude=n10]" mpirun -srun ./hello_world

```
Job <1892> is submitted to default queue <interactive>.
<<Waiting for dispatch ...>>
<<Starting on lsfhost.localdomain>>
Hello world! I'm 0 of 8 on n8
Hello world! I'm 1 of 8 on n8
Hello world! I'm 6 of 8 on n12
Hello world! I'm 2 of 8 on n9
Hello world! I'm 4 of 8 on n11
Hello world! I'm 7 of 8 on n12
Hello world! I'm 3 of 8 on n9
Hello world! I'm 5 of 8 on n11
```

  In addition to displaying interconnect selection information, the `mpirun`-prot option can be used to verify that application ranks have been allocated in the required manner:

  % bsub -I -n12 $MPI_ROOT/bin/mpirun -prot -srun -n6 -N6 ./a.out

```
Job <1472> is submitted to default queue <interactive>.
<<Waiting for dispatch ...>>
<<Starting on lsfhost.localdomain>>
Host 0 -- ip 172.20.0.8 -- ranks 0
Host 1 -- ip 172.20.0.9 -- ranks 1
Host 2 -- ip 172.20.0.10 -- ranks 2
Host 3 -- ip 172.20.0.11 -- ranks 3
Host 4 -- ip 172.20.0.12 -- ranks 4
Host 5 -- ip 172.20.0.13 -- ranks 5
host | 0  1  2  3  4  5
======|============================
0 : SHM VAPI VAPI VAPI VAPI VAPI
1 : VAPI SHM VAPI VAPI VAPI VAPI
2 : VAPI VAPI SHM VAPI VAPI VAPI
3 : VAPI VAPI VAPI SHM VAPI VAPI
4 : VAPI VAPI VAPI VAPI SHM VAPI
```

```
5 : VAPI VAPI VAPI VAPI VAPI SHM
Hello world! I'm 0 of 6 on n8
Hello world! I'm 3 of 6 on n11
Hello world! I'm 5 of 6 on n13
Hello world! I'm 4 of 6 on n12
Hello world! I'm 2 of 6 on n10
Hello world! I'm 1 of 6 on n9
```

- Use LSF on non-HP XC systems

  On non-HP XC systems, to invoke the Parallel Application Manager (PAM) feature of LSF for applications where all processes execute the same program on the same host:

  % **bsub <*lsf_options*> pam -mpi mpirun <*mpirun_options*> *program* <*args*>**

  In this case, LSF assigns a host to the MPI job.

  For example:

  % bsub pam -mpi $MPI_ROOT/bin/mpirun -np 4 compute_pi

  requests a host assignment from LSF and runs the `compute_pi` application with four processes.

  The load-sharing facility (LSF) allocates hosts to run an MPI job. In general, LSF improves resource usage for MPI jobs that run in multihost environments. LSF handles the job scheduling and the allocation of the necessary hosts and Platform MPI handles the task of starting the application's processes on the hosts selected by LSF.

  By default `mpirun` starts the MPI processes on the hosts specified by the user, in effect handling the direct mapping of host names to IP addresses. When you use LSF to start MPI applications, the host names, specified to `mpirun` or implicit when the -h option is not used, are treated as symbolic variables that refer to the IP addresses that LSF assigns. Use LSF to do this mapping by specifying a variant of `mpirun` to execute your job.

  To invoke LSF for applications that run on multiple hosts:

  % **bsub [*lsf_options*] pam -mpi mpirun [*mpirun_options*] -f appfile [-- *extra_args_for_appfile*]**

  In this case, each host specified in the appfile is treated as a symbolic name, referring to the host that LSF assigns to the MPI job.

  For example:

  % **bsub pam -mpi $MPI_ROOT/bin/mpirun -f *my_appfile***

  runs an appfile named my_appfile and requests host assignments for all remote and local hosts specified in my_appfile. If my_appfile contains the following items:

  ```
  -h voyager -np 10 send_receive
  -h enterprise -np 8 compute_pi
  ```

  Host assignments are returned for the two symbolic links voyager and enterprise.

  When requesting a host from LSF, be sure that the path to your executable file is accessible by all machines in the resource pool.

More information about appfile runs

This example teaches you how to run the `hello_world.c` application that you built on HP and Linux (above) using two hosts to achieve four-way parallelism. For this example, the local host is named jawbone and a remote host is named wizard. To run `hello_world.c` on two hosts, use the following procedure, replacing jawbone and wizard with the names of your machines.

1. Edit the `.rhosts` file on hosts
   ```
   jawbone
   ```

and
`wizard`
.

Add an entry for
`wizard`
in the `.rhosts` file on
`jawbone`
and an entry for
`jawbone`
in the `.rhosts` file on
`wizard`
. In addition to the entries in the `.rhosts` file, be sure the correct commands and permissions are set up in the ssh shell configuration file on all hosts so you can start your remote processes.

2. Configure ssh so you can ssh into the machines without typing a password.
3. Be sure the executable is accessible from each host by placing it in a shared directory or by copying it to a local directory on each host.
4. Create an appfile.

   An appfile is a text file that contains process counts and a list of programs. In this example, create an appfile named `my_appfile` containing the following lines:

   ```
   -h jawbone -np 2 /path/to/hello_world
   -h wizard -np 2 /path/to/hello_world
   ```

   The appfile file should contain a separate line for each host. Each line specifies the name of the executable file and the number of processes to run on the host. The -h option is followed by the name of the host where the specified processes must be run. Instead of using the host name, you can use its IP address.
5. Run the `hello_world` executable file:

   % **$MPI_ROOT/bin/mpirun -f *my_appfile***

   The -f option specifies the file name that follows it is an appfile. `mpirun` parses the appfile, line by line, for the information to run the program.

   In this example, `mpirun` runs the hello_world program with two processes on the local machine, jawbone, and two processes on the remote machine, wizard, as dictated by the -np 2 option on each line of the appfile.
6. Analyze hello_world output.

   HP-MPI prints the output from running the hello_world executable in nondeterministic order.

   The following is an example of the output:

   ```
   Hello world! I’m 2 of 4 on wizard
   Hello world! I’m 0 of 4 on jawbone
   Hello world! I’m 3 of 4 on wizard
   Hello world! I’m 1 of 4 on jawbone
   ```

   Processes 0 and 1 run on jawbone, the local host, while processes 2 and 3 run on wizard. HP-MPI guarantees that the ranks of the processes in *MPI_COMM_WORLD* are assigned and sequentially ordered according to the order the programs appear in the appfile. The appfile in this example, my_appfile, describes the local host on the first line and the remote host on the second line.

Running MPMD applications

A multiple program multiple data (MPMD) application uses two or more programs to functionally decompose a problem. This style can be used to simplify the application source and reduce the size of spawned processes. Each process can execute a different program.

MPMD with appfiles

To run an MPMD application, the `mpirun` command must reference an appfile that contains the list of programs to be run and the number of processes to be created for each program.

A simple invocation of an MPMD application looks like this:

% **$MPI_ROOT/bin/mpirun -f *appfile***

where *appfile* is the text file parsed by `mpirun` and contains a list of programs and process counts.

Suppose you decompose the poisson application into two source files: poisson_master (uses a single master process) and poisson_child (uses four child processes).

The appfile for the example application contains the two lines shown below:

```
-np 1 poisson_master

-np 4 poisson_child
```

To build and run the example application, use the following command sequence:

% **$MPI_ROOT/bin/mpicc -o poisson_master poisson_master.c**

% **$MPI_ROOT/bin/mpicc -o poisson_child poisson_child.c**

% **$MPI_ROOT/bin/mpirun -f *appfile***

MPMD with prun

`prun` also supports running applications with MPMD using procfiles. See the `prun` documentation at *http://www.quadrics.com*.

MPMD with srun

MPMD is not directly supported with `srun`. However, users can write custom wrapper scripts to their application to emulate this functionality. This can be accomplished by using the environment variables *SLURM_PROCID* and *SLURM_NPROCS* as keys to selecting the correct executable.

Modules on Linux

Modules are a convenient tool for managing environment settings for packages. Platform MPI for Linux provides an `Platform MPI` module at `/opt/platform_mpi/modulefiles/platform-mpi`, which sets *MPI_ROOT* and adds to *PATH* and *MANPATH*. To use it, copy the file to a system-wide module directory, or append `/opt/platform_mpi/modulefiles/platform-mpi` to the *MODULEPATH* environment variable.

Some useful module-related commands are:

module avail

Lists modules that can be loaded

module load platform-mpi

Loads the Platform MPI module

module list

Lists loaded modules

module unload platform-mpi

Unloads the Platform MPI module

Modules are only supported on Linux.

Note:

On HP XC Linux, the Platform MPI module is named `mpi/hp/default` and can be abbreviated as '`mpi`'.

Run-time utility commands

Platform MPI provides a set of utility commands to supplement MPI library routines.

mpirun

This section includes a discussion of `mpirun` syntax formats, `mpirun` options, appfiles, the multipurpose daemon process, and generating multihost instrumentation profiles.

The Platform MPI start-up `mpirun` requires that MPI be installed in the same directory on every execution host. The default is the location where `mpirun` is executed. This can be overridden with the *MPI_ROOT* environment variable. Set the *MPI_ROOT* environment variable prior to starting `mpirun`.

`mpirun` syntax has six formats:

- Single host execution
- Appfile execution
- `prun` execution
- `srun` execution
- LSF on HP XC systems
- LSF on non-HP XC systems

Single host execution

- To run on a single host, you can use the -np option to `mpirun`.

  For example:

  % **$MPI_ROOT/bin/mpirun -np 4 ./a.out**

  will run 4 ranks on the local host.

Appfile execution

- For applications that consist of multiple programs or that run on multiple hosts, here is a list of common options. For a complete list, see the `mpirun` manpage:

  % **mpirun [-help] [-version] [-djpv] [-ck] [-t *spec*] [-i *spec*] [-commd] [-tv] -f *appfile* [--*extra_args_for_appfile*]**

  Where --*extra_args_for_appfile* specifies extra arguments to be applied to the programs listed in the appfile. This is a space-separated list of arguments. Use this option at the end of a command line to append extra arguments to each line of your appfile. These extra arguments also apply to spawned applications if specified on the `mpirun` command line.

  In this case, each program in the application is listed in a file called an appfile.

For example:

% **$MPI_ROOT/bin/mpirun -f *my_appfile***

runs using an appfile named my_appfile, that might have contents such as:

```
-h hostA -np 2 /path/to/a.out
-h hostB -np 2 /path/to/a.out
```

which specify that two ranks are to run on host A and two on host B.

prun execution

- Use the -prun option for applications that run on the Quadrics Elan interconnect. When using the -prun option, `mpirun` sets environment variables and invokes `prun` utilities.

  The -prun argument to `mpirun` specifies that the `prun` command is to be used for launching. All arguments following -prun are passed unmodified to the `prun` command.

  % **$MPI_ROOT/bin/mpirun <*mpirun options*> -prun <*prun options*>**

  The -np option is not allowed with `prun`. Some features like `mpirun -stdio` processing are unavailable.

  % **$MPI_ROOT/bin/mpirun -prun -n 2 ./a.out**

  launches `a.out` on two processors.

  % **$MPI_ROOT/bin/mpirun -prot -prun -n 6 -N 6 ./a.out**

  turns on the print protocol option (-prot is an `mpirun` option, and therefore is listed before -prun) and runs on 6 machines, one CPU per node.

  Platform MPI also provides implied `prun` mode. The implied `prun` mode allows the user to omit the -prun argument from the `mpirun` command line with the use of the environment variable *MPI_USEPRUN*.

srun execution

- Applications that run on HP XC clusters require the -srun option. start-up directly from `srun` is not supported. When using this option, `mpirun` sets environment variables and invokes `srun` utilities.

  The -srun argument to `mpirun` specifies that the `srun` command is to be used for launching. All arguments following -srun are passed unmodified to the `srun` command.

  % **$MPI_ROOT/bin/mpirun <*mpirun options*> -srun <*srun options*>**

  The -np option is not allowed with `srun`. Some features like `mpirun -stdio` processing are unavailable.

  % **$MPI_ROOT/bin/mpirun -srun -n 2 ./a.out**

  launches `a.out` on two processors.

  % **$MPI_ROOT/bin/mpirun -prot -srun -n 6 -N 6 ./a.out**

  turns on the print protocol option (-prot is an `mpirun` option, and therefore is listed before -srun) and runs on 6 machines, one CPU per node.

Platform MPI also provides implied `srun` mode. The implied `srun` mode allows the user to omit the -srun argument from the `mpirun` command line with the use of the environment variable *MPI_USESRUN*.

LSF on HP XC Systems

Platform MPI jobs can be submitted using LSF. LSF uses the SLURM `srun` launching mechanism. Because of this, Platform MPI jobs must specify the -srun option whether LSF is used or `srun` is used.

```
% bsub -I -n2 $MPI_ROOT/bin/mpirun -srun ./a.out
```

LSF on Non-HP XC Systems

On non-HP XC systems, to invoke the Parallel Application Manager (PAM) feature of LSF for applications where all processes execute the same program on the same host:

```
% bsub <lsf_options> pam -mpi mpirun <mpirun_options> program <args>
```

Appfiles

An appfile is a text file that contains process counts and a list of programs. When you invoke `mpirun` with the name of the appfile, `mpirun` parses the appfile to get information for the run.

Creating an appfile

The format of entries in an appfile is line oriented. Lines that end with the backslash (\) character are continued on the next line, forming a single logical line. A logical line starting with the pound (#) character is treated as a comment. Each program, along with its arguments, is listed on a separate logical line.

The general form of an appfile entry is:

```
[-h remote_host] [-e var[=val] [...]] [-sp paths] [-np #] program [args]
```

where

-h *remote_host*

Specifies the remote host where a remote executable file is stored. The default is to search the local host. *remote_host* is a host name or an IP address.

-e *var=val*

Sets the environment variable *var* for the program and gives it the value *val*. The default is not to set environment variables. When you use -e with the -h option, the environment variable is set to *val* on the remote host.

-sp *paths*

Sets the target shell *PATH* environment variable to *paths*. Search paths are separated by a colon. Both -sp path and -e PATH=path do the same thing. If both are specified, the -e PATH=path setting is used.

-np #

Specifies the number of processes to run. The default value for # is 1.

program

Specifies the name of the executable to run. `mpirun` searches for the executable in the paths defined in the *PATH* environment variable.

args

Specifies command-line arguments to the program. Options following a program name in your appfile are treated as program arguments and are not processed by `mpirun`.

Adding program arguments to your appfile

When you invoke `mpirun` using an appfile, arguments for your program are supplied on each line of your appfile. Platform MPI also provides an option on your `mpirun` command line to provide additional program arguments to those in your appfile. This is useful if you want to specify extra arguments for each program listed in your appfile, but do not want to edit your appfile.

To use an appfile when you invoke `mpirun`, use one of the following:

- `mpirun [`*`mpirun_options`*`] -f `*`appfile`*` [--`*`extra_args_for_appfile`*`]`
- `bsub [`*`lsf_options`*`] pam -mpi mpirun [`*`mpirun_options`*`] -f `*`appfile`*` [--`*`extra_args_for_appfile`*`]`

The `--` *extra_args_for_appfile* option is placed at the end of your command line, after *appfile*, to add options to each line of your appfile.

Caution:

Arguments placed after -- are treated as program arguments, and are not processed by `mpirun`. Use this option when you want to specify program arguments for each line of the appfile, but want to avoid editing the appfile.

For example, suppose your appfile contains

```
-h voyager -np 10 send_receive arg1 arg2
-h enterprise -np 8 compute_pi
```

If you invoke `mpirun` using the following command line:

`mpirun -f `*`appfile -- arg3 - arg4 arg5`*

- The `send_receive` command line for machine voyager becomes:

  `send_receive arg1 arg2 arg3 -arg4 arg5`
- The `compute_pi` command line for machine enterprise becomes:

  `compute_pi arg3 -arg4 arg5`

When you use the `--` *extra_args_for_appfile* option, it must be specified at the end of the `mpirun` command line.

Setting remote environment variables

To set environment variables on remote hosts use the -e option in the appfile. For example, to set the variable *MPI_FLAGS*:

`-h` *remote_host* `-e MPI_FLAGS=`*val* `[-np #]` *program* `[`*args*`]`

Assigning ranks and improving communication

The ranks of the processes in *MPI_COMM_WORLD* are assigned and sequentially ordered according to the order the programs appear in the appfile.

For example, if your appfile contains

```
-h voyager -np 10 send_receive
-h enterprise -np 8 compute_pi
```

Platform MPI assigns ranks 0 through 9 to the 10 processes running send_receive and ranks 10 through 17 to the 8 processes running `compute_pi`.

You can use this sequential ordering of process ranks to your advantage when you optimize for performance on multihost systems. You can split process groups according to communication patterns to reduce or remove interhost communication hot spots.

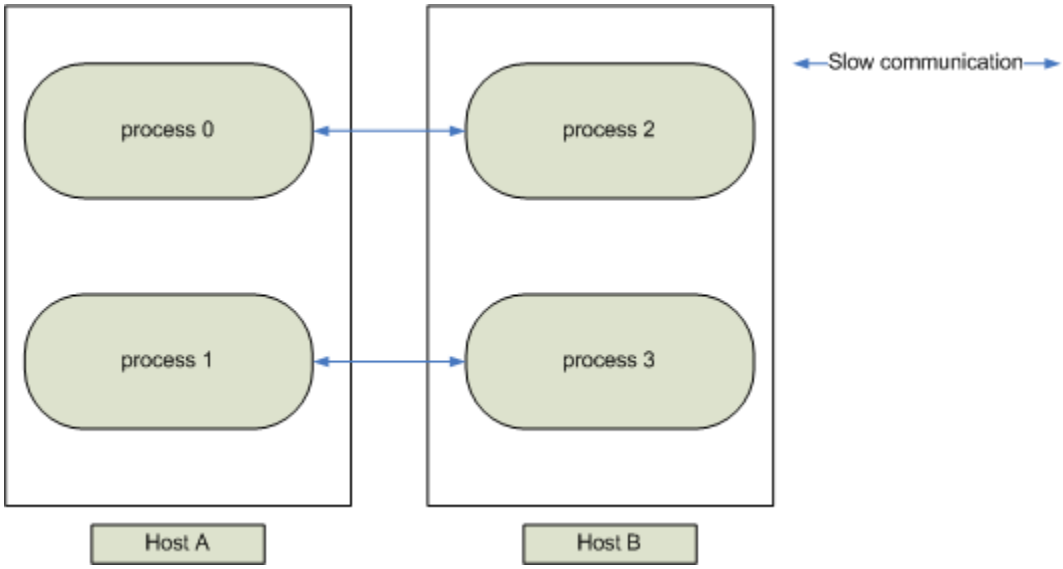
For example, if you have the following:

- A multihost run of four processes
- Two processes per host on two hosts
- Higher communication traffic between ranks 0: 2 and 1: 3

You could use an appfile that contains the following:

```
-h hosta -np 2 program1
-h hostb -np 2 program2
```

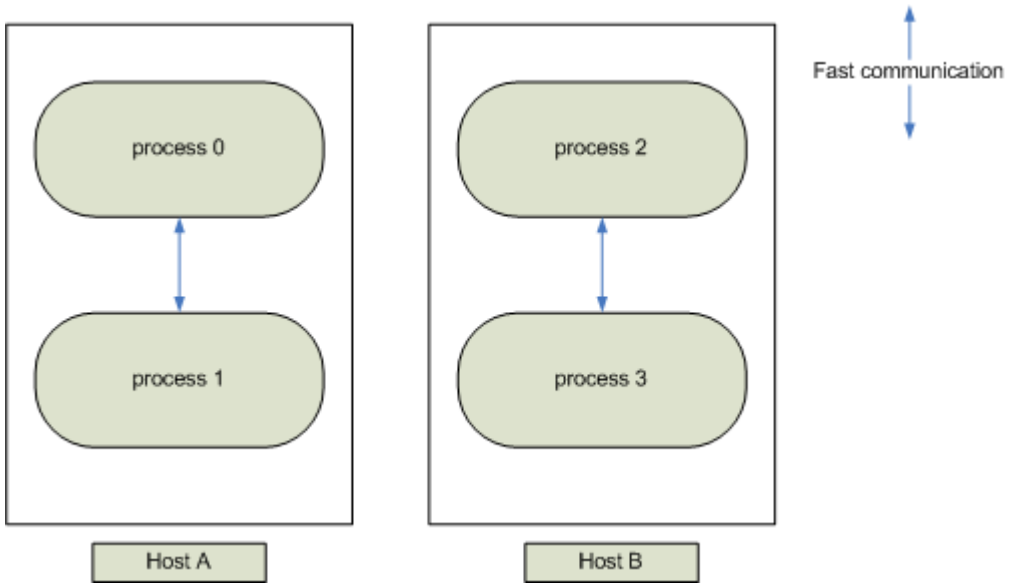
However, this places processes 0 and 1 on host a and processes 2 and 3 on host b, resulting in interhost communication between the ranks identified as having slow communication.



A more optimal appfile for this example would be:

```
-h hosta -np 1 program1
-h hostb -np 1 program2
-h hosta -np 1 program1
-h hostb -np 1 program2
```

This places ranks 0 and 2 on host a and ranks 1 and 3 on host b. This placement allows intrahost communication between ranks that are identified as communication hot spots. Intrahost communication yields better performance than interhost communication.



Multipurpose daemon process

Platform MPI incorporates a multipurpose daemon process that provides start-up, communication, and termination services. The daemon operation is transparent. Platform MPI sets up one daemon per host (or appfile entry) for communication.

Note:

Because Platform MPI sets up one daemon per host (or appfile entry) for communication, when you invoke your application with -np x, Platform MPI generates x+1 processes.

Generating multihost instrumentation profiles

When you enable instrumentation for multihost runs, and invoke `mpirun` on a host where at least one MPI process is running, or on a host remote from MPI processes, Platform MPI writes the instrumentation output file (*prefix*.instr) to the working directory on the host that is running rank 0 (when instrumentation for multihost runs is enabled). When using -ha, the output file is located on the host that is running the lowest existing rank number at the time the instrumentation data is gathered during MPI_FINALIZE()

mpiexec

The MPI-2 standard defines `mpiexec` as a simple method to start MPI applications. It supports fewer features than `mpirun`, but it is portable. `mpiexec` syntax has three formats:

- `mpiexec` offers arguments similar to a MPI_Comm_spawn call, with arguments as shown in the following form:

  `mpiexec [-n` *`maxprocs`*`][-soft` *`ranges`*`][-host` *`host`*`][-arch` *`arch`*`][-wdir` *`dir`*`][-path` *`dirs`*`][-file` *`file`*`]`*`command-args`*

  For example:

  % **$MPI_ROOT/bin/mpiexec -n 8 ./myprog.x 1 2 3**

creates an 8 rank MPI job on the local host consisting of 8 copies of the program myprog.x, each with the command-line arguments 1, 2, and 3.

- It also allows arguments like a MPI_Comm_spawn_multiple call, with a colon-separated list of arguments, where each component is like the form above.

  For example:

  ```
  % $MPI_ROOT/bin/mpiexec -n 4 ./myprog.x : -host host2 -n 4 /path/to/myprog.x
  ```

  creates a MPI job with 4 ranks on the local host and 4 on host 2.

- Finally, the third form allows the user to specify a file containing lines of data like the arguments in the first form.

  ```
  mpiexec [-configfile file]
  ```

  For example:

  ```
  % $MPI_ROOT/bin/mpiexec -configfile cfile
  ```

  gives the same results as in the second example, but using the -configfile option (assuming the file cfile contains -n 4 ./myprog.x -host host2 -n 4 -wdir /some/path ./myprog.x)

where `mpiexec` options are:

-n *maxprocs*

Creates *maxprocs* MPI ranks on the specified host.

-soft *range-list*

Ignored in Platform MPI.

-host *host*

Specifies the host on which to start the ranks.

-arch *arch*

Ignored in Platform MPI.

-wdir *dir*

Specifies the working directory for the created ranks.

-path *dirs*

Specifies the *PATH* environment variable for the created ranks.

-file *file*

Ignored in Platform MPI.

This last option is used separately from the options above.

-configfile *file*

Specify a file of lines containing the above options.

`mpiexec` does not support `prun` or `srun` start-up.

mpijob

`mpijob` lists the Platform MPI jobs running on the system. `mpijob` can only be used for jobs started in appfile mode. Invoke `mpijob` on the same host as you initiated `mpirun`. The `mpijob` syntax is:

```
mpijob [-help] [-a] [-u] [-j id] [id id ...]]
```

where

-help

Prints usage information for the utility.

-a

Lists jobs for all users.

-u

Sorts jobs by user name.

-j *id*

Provides process status for job *id*. You can list a number of job IDs in a space-separated list.

When you invoke `mpijob`, it reports the following information for each job:

JOB

Platform MPI job identifier.

USER

User name of the owner.

NPROCS

Number of processes.

PROGNAME

Program names used in the Platform MPI application.

By default, your jobs are listed by job ID in increasing order. However, you can specify the -a and -u options to change the default behavior.

An `mpijob` output using the -a and -u options is shown below, listing jobs for all users and sorting them by user name.

```
JOB USER NPROCS PROGNAME
22623 charlie 12 /home/watts
22573 keith  14 /home/richards
22617 mick 100 /home/jagger
22677     ron 4 /home/wood
```

When you specify the -j option, `mpijob` reports the following for each job:

RANK

Rank for each process in the job.

HOST

Host where the job is running.

PID

Process identifier for each process in the job.

LIVE

Whether the process is running (an x is used) or has been terminated.

PROGNAME

Program names used in the Platform MPI application.

`mpijob` does not support `prun` or `srun` start-up.

`mpijob` is not available on Platform MPI V1.0 for Windows.

mpiclean

`mpiclean` kills processes in Platform MPI applications started in appfile mode. Invoke mpiclean on the host where you initiated mpirun.The MPI library checks for abnormal termination of processes while your application is running. In some cases, application bugs can cause processes to deadlock and linger in the system. When this occurs, you can use mpijob to identify hung jobs and `mpiclean` to kill all processes in the hung application.

`mpiclean` syntax has two forms:

1. `mpiclean [-help] [-v] -j id [id id ....]`
2. `mpiclean [-help] [-v] -m`

where

-help

Prints usage information for the utility.

-v

Turns on verbose mode.

-m

Cleans up shared-memory segments.

-j *id*

Kills the processes of job number ID. You can specify multiple job IDs in a space-separated list. Obtain the job ID using the -j option when you invoke mpirun.

You can only kill jobs that are your own.

The second syntax is used when an application aborts during MPI_Init, and the termination of processes does not destroy the allocated shared-memory segments.

`mpiclean` does not support `prun` or `srun` start-up.

`mpiclean` is not available on Platform MPI V1.0 for Windows.

Interconnect support

Platform MPI supports a variety of high-speed interconnects. Platform MPI attempts to identify and use the fastest available high-speed interconnect by default.

The search order for the interconnect is determined by the environment variable *MPI_IC_ORDER* (which is a colon-separated list of interconnect names), and by command-line options which take higher precedence.

Table 15: Interconnect command-line options

Command-Line Option	Protocol Specified	OS
-ibv / -IBV		Linux
-vapi / -VAPI	VAPI: Mellanox Verbs API	Linux
-udapl / -UDAPL	uDAPL: InfiniBand and some others	Linux
-psm / -PSM	PSM: QLogic InfiniBand	Linux
-mx / -MX	MX: Myrinet	• Linux • Windows
-gm / -GM	GM: Myrinet	Linux
-elan / -ELAN	Quadrics Elan3 or Elan4	Linux
-itapi / -ITAPI	ITAPI: InfiniBand	Linux (deprecated)
-ibal / -IBAL	IBAL: Windows IB Access Layer	Windows
-TCP	TCP/IP	All

The interconnect names used in *MPI_IC_ORDER* are like the command-line options above, but without the dash. On Linux, the default value of *MPI_IC_ORDER* is

```
psm:ibv:vapi:udapl:itapi:mx:gm:elan:tcp
```

If command-line options from the above table are used, the effect is that the specified setting is implicitly prepended to the *MPI_IC_ORDER* list, taking higher precedence in the search.

The availability of an interconnect is determined based on whether the relevant libraries can use `dlopen` / `shl_load`, and on whether a recognized module is loaded in Linux. If either condition is not met, the interconnect is determined to be unavailable.

Interconnects specified in the command line or in the *MPI_IC_ORDER* variable can be lower case or upper case. Lower case means the interconnect is used if available. Upper case options are handled slightly differently between Linux and Windows. On Linux, the upper case option instructs Platform MPI to abort if the specified interconnect is determined to be unavailable by the interconnect detection process. On Windows, the upper case option instructs HP-MPI to ignore the results of interconnect detection and simply try to run using the specified interconnect irrespective of whether it appears to be available or not.

On Linux, the names and locations of the libraries to be opened, and the names of the recognized interconnect module names are specified by a collection of environment variables that are in `$MPI_ROOT/etc/pcmpi.conf`.

The `pcmpi.conf` file can be used for any environment variables, but arguably its most important use is to consolidate environment variables related to interconnect selection.

The default value of *MPI_IC_ORDER* is specified there, along with a collection of variables of the form:

MPI_ICLIB_XXX__YYY

MPI_ICMOD_XXX__YYY

where *XXX* is one of the interconnects (IBV, VAPI, etc.) and *YYY* is an arbitrary suffix. The *MPI_ICLIB_** variables specify names of libraries to be called by `dlopen`. The *MPI_ICMOD_** variables specify regular expressions for names of modules to search for.

An example is the following two pairs of variables for PSM:

MPI_ICLIB_PSM__PSM_MAIN = libpsm_infinipath.so.1

MPI_ICMOD_PSM__PSM_MAIN="^ib_ipath "

and

MPI_ICLIB_PSM__PSM_PATH = /usr/lib64/libpsm_infinipath.so.1

MPI_ICMOD_PSM__PSM_PATH="^ib_ipath "

The suffixes *PSM_MAIN* and *PSM_PATH* are arbitrary and represent two attempts that are made when determining if the PSM interconnect is available.

The list of suffixes is in the variable *MPI_IC_SUFFIXES*, which is also set in the `pcmpi.conf` file.

So, when Platform MPI is determining the availability of the PSM interconnect, it first looks at:

MPI_ICLIB_PSM__PSM_MAIN

MPI_ICMOD_PSM__PSM_MAIN

for the library to use `dlopen` and the module name to look for. Then, if that fails, it continues on to the next pair:

MPI_ICLIB_PSM__PSM_PATH

MPI_ICMOD_PSM__PSM_PATH

which, in this case, specifies a full path to the PSM library.

The *MPI_ICMOD_** variables allow relatively complex values to specify the module names to be considered as evidence that the specified interconnect is available. Consider the example:

MPI_ICMOD_VAPI__VAPI_MAIN="^mod_vapi " || "^mod_vip " || "^ib_core "

This means any of those three names will be accepted as evidence that VAPI is available. Each of those strings is searched for individually in the output from `/sbin/lsmod`. The carrot in the search pattern is used to signify the beginning of a line, but the rest of regular expression syntax is not supported.

In many cases, if a system has a high-speed interconnect that is not found by Platform MPI due to changes in library names and locations or module names, the problem can be fixed by simple edits to the `pcmpi.conf` file. Contacting Platform MPI support for assistance is encouraged.

Protocol-specific options and information

This section briefly describes the available interconnects and illustrates some of the more frequently used interconnects options.

TCP/IP

TCP/IP is supported on many types of cards. Machines often have more than one IP address, and a user can specify the interface to be used to get the best performance.

Platform MPI does not inherently know which IP address corresponds to the fastest available interconnect card. By default IP addresses are selected based on the list returned by gethostbyname(). The `mpirun` option -netaddr can be used to gain more explicit control over which interface is used.

IBAL

IBAL is only supported on Windows. Lazy deregistration is not supported with IBAL.

IBV

Platform MPI supports OpenFabrics Enterprise Distribution (OFED) through V1.4. Platform MPI can use both the verbs 1.0 and 1.1 interface.

To use OFED on Linux, the memory size for locking must be specified (see below). It is controlled by the `/etc/security/limits.conf` file for Red Hat and the `/etc/syscntl.conf` file for SuSE.

```
* soft memlock 4194303

* hard memlock 4194304
```

The example above uses the maximum locked-in-memory address space in KB units. The recommendation is to set the value to half of the physical memory on the machine. Platform MPI tries to pin up to 20% of the machine's memory (see MPI_PHYSICAL_MEMORY and MPI_PIN_PERCENTAGE) and fails if it is unable to pin the desired amount of memory.

Machines can have multiple InfiniBand cards. By default each Platform MPI rank selects one card for its communication and the ranks cycle through the available cards on the system, so the first rank uses the first card, the second rank uses the second card, etc.

The environment variable *MPI_IB_CARD_ORDER* can be used to control which card the ranks select. Or, for increased potential bandwidth and greater traffic balance between cards, each rank can be instructed to use multiple cards by using the variable *MPI_IB_MULTIRAIL*.

Lazy deregistration is a performance enhancement used by Platform MPI on several high speed interconnects on Linux. This option is turned on by default, and requires the application to be linked in such a way that Platform MPI can intercept calls to `malloc`, `munmap`, etc. Most applications are linked that way, but if one is not then Platform MPI's lazy deregistration can be turned off with the command line -ndd.

Some applications decline to directly link to `libmpi` and instead link to a wrapper library that is linked to `libmpi`. In this case it is still possible for Platform MPI's `malloc` etc. interception to be used by supplying the --auxiliary option to the linker when creating the wrapper library, by using a compiler flag such as -Wl, --auxiliary, libmpi.so.

Dynamic linking is required with all InfiniBand use on Linux.

Platform MPI does not use the Connection Manager (CM) library with OFED.

InfiniBand card failover

When InfiniBand has multiple paths or connections to the same node, Platform MPI supports InfiniBand card failover. This functionality is always enabled. An InfiniBand connection is setup between every card-pair. During normal operation, short messages are alternated among the connections in round-robin manner. Long messages are striped over all the connections. When one of the connections is broken, a warning is issued, but Platform MPI continues to use the rest of the healthy connections to transfer messages. If all the connections are broken, Platform MPI issues an error message.

InfiniBand port failover

A multi-port InfiniBand channel adapter can use automatic path migration (APM) to provide network high availability. APM is defined by the InfiniBand Architecture Specification, and enables Platform MPI to recover from network failures by specifying and using the alternate paths in the network. The InfiniBand subnet manager defines one of the server's links as primary and one as redundant/alternate. When the primary link fails, the channel adapter automatically redirects traffic to the redundant path when a link failure is detected. This support is provided by the InfiniBand driver available in OFED 1.2 and later

releases. Redirection and reissued communications are performed transparently to applications running on the cluster.

The user has to explicitly enable APM by setting the environment variable *MPI_HA_NW_PORT_FAILOVER*=1, as in the following example:

```
% /opt/platform_mpi/bin/mpirun -np 4 -prot -e MPI_HA_NW_PORT_FAILOVER=1 -hostlist
nodea,nodeb,nodec,noded /my/dir/hello_world
```

When the *MPI_HA_NW_PORT_FAILOVER* environment variable is set, Platform MPI identifies and specifies the primary and the alternate paths (if available) when it sets up the communication channels between the ranks. It also requests the InfiniBand driver to load the alternate path for a potential path migration if a network failure occurs. When a network failure occurs, the InfiniBand driver automatically transitions to the alternate path, notifies Platform MPI of the path migration, and continues the network communication on the alternate path. At this point, Platform MPI also reloads the original primary path as the new alternate path. If this new alternate path is restored, this will allow for the InfiniBand driver to automatically migrate to it in case of future failures on the new primary path. However, if the new alternate path is not restored, or if alternate paths are unavailable on the same card, future failures will force Platform MPI to try to failover to alternate cards if available. All of these operations are performed transparent to the application that uses Platform MPI.

If the environment has multiple cards, with multiple ports per card, and has APM enabled, Platform MPI gives InfinBand port failover priority over card failover.

InfiniBand with MPI_Comm_connect() and MPI_Comm_accept()

Platform MPI supports MPI_Comm_connect() and MPI_Comm_accept() over InfiniBand processes using the IBV protocol. Both sides must have InfiniBand support enabled and use the same InfiniBand parameter settings.

MPI_Comm_connect() and MPI_Comm_accept() need a port name, which is the IP and port at the root process of the accept side. First, a TCP connection is established between the root process of both sides. Next, TCP connections are setup among all the processes. Finally, IBV InfiniBand connections are established among all process-pairs and the TCP connections are closed.

VAPI

The *MPI_IB_CARD_ORDER* card selection option and the -ndd option described above for IBV applies to VAPI.

uDAPL

The -ndd option described above for IBV applies to uDAPL.

GM

The -ndd option described above for IBV applies to GM.

Elan

Platform MPI supports the Elan3 and Elan4 protocols for Quadrics.

By default Platform MPI uses Elan collectives for broadcast and barrier. If messages are outstanding at the time the Elan collective is entered and the other side of the message enters a completion routine on the outstanding message before entering the collective call, it is possible for the application to hang due to lack of message progression while inside the Elan collective. This is an uncommon situation in real

applications. If such hangs are observed, disable Elan collectives using the environment variable *MPI_USE_LIBELAN=0*.

Interconnect selection examples

The default *MPI_IC_ORDER* generally results in the fastest available protocol being used. The following example uses the default ordering and supplies a -netaddr setting, in case TCP/IP is the only interconnect available.

```
% echo MPI_IC_ORDER
ibv:vapi:udapl:psm:mx:gm:elan:tcp
% export MPIRUN_SYSTEM_OPTIONS="-netaddr 192.168.1.0/24"
% export MPIRUN_OPTIONS="-prot"
% $MPI_ROOT/bin/mpirun -srun -n4 ./a.out
```

The command line for the above appears to `mpirun` as `$MPI_ROOT/bin/mpirun -netaddr 192.168.1.0/24 -prot -srun -n4 ./a.out` and the interconnect decision looks for IBV, then VAPI, etc. down to TCP/IP. If TCP/IP is chosen, it uses the 192.168.1.* subnet.

If TCP/IP is needed on a machine where other protocols are available, the -TCP option can be used.

This example is like the previous, except TCP is searched for and found first. (TCP should always be available.) So TCP/IP is used instead of IBV or Elan, etc.

```
% $MPI_ROOT/bin/mpirun -TCP -srun -n4 ./a.out
```

The following example output shows three runs on an Elan system, first using Elan as the protocol, then using TCP/IP over GigE, then using TCP/IP over the Quadrics card.

- This runs on Elan

  ```
  [user@opte10 user]$ bsub -I -n3 -ext "SLURM[nodes=3]" $MPI_ROOT/bin/mpirun
  -prot -srun ./a.out Job <59304> is submitted to default queue <normal>.
  <<Waiting for dispatch ...>> <<Starting on lsfhost.localdomain>> Host 0 --
  ELAN node 0 -- ranks 0 Host 1 -- ELAN node 1 -- ranks 1 Host 2 -- ELAN node
  2 -- ranks 2  host | 0 1 2 ======|================ 0 : SHM ELAN ELAN 1 :
  ELAN SHM ELAN 2 : ELAN ELAN SHM Hello world! I'm 0 of 3 on opte6 Hello
  world! I'm 1 of 3 on opte7 Hello world! I'm 2 of 3 on opte8
  ```

- This runs on TCP/IP over the GigE network configured as 172.20.x.x on eth0

  ```
  [user@opte10 user]$ bsub -I -n3 -ext "SLURM[nodes=3]" $MPI_ROOT/bin/mpirun
  -prot -TCP -srun ./a.out Job <59305> is submitted to default queue
  <normal>. <<Waiting for dispatch ...>> <<Starting on lsfhost.localdomain>>
  Host 0 -- ip 172.20.0.6 -- ranks 0 Host 1 -- ip 172.20.0.7 -- ranks 1 Host
  2 -- ip 172.20.0.8 -- ranks 2  host | 0 1 2 ======|================ 0 : SHM
  TCP TCP 1 : TCP SHM TCP 2 : TCP TCP SHMHello world! I'm 0 of 3 on opte6 Hello
  world! I'm 1 of 3 on opte7 Hello world! I'm 2 of 3 on opte8
  ```

- This uses TCP/IP over the Elan subnet using the -TCP option with the -netaddr option for the Elan interface 172.22.x.x

  ```
  [user@opte10 user]$ bsub -I -n3 -ext "SLURM[nodes=3]" $MPI_ROOT/bin/mpirun
  -prot -TCP -netaddr 172.22.0.10 -srun ./a.out Job <59307> is submitted to
  default queue <normal>. <<Waiting for dispatch ...>> <<Starting on
  lsfhost.localdomain>> Host 0 -- ip 172.22.0.2 -- ranks 0 Host 1 -- ip
  172.22.0.3 -- ranks 1 Host 2 -- ip 172.22.0.4 -- ranks 2 host | 0 1 2
  ```

```
======|================ 0 : SHM TCP TCP 1 : TCP SHM TCP 2 : TCP TCP SHMHello
world! I'm 0 of 3 on opte2 Hello world! I'm 1 of 3 on opte3 Hello world! I'm
2 of 3 on opte4
```

- Elan interface

```
[user@opte10 user]$ /sbin/ifconfig eip0 eip0 Link encap:Ethernet HWaddr
00:00:00:00:00:0F       inet addr:172.22.0.10 Bcast:172.22.255.255 Mask:
255.255.0.0      UP BROADCAST RUNNING MULTICAST MTU:65264 Metric:1      RX
packets:38 errors:0 dropped:0 overruns:0 frame:0      TX packets:6 errors:
0 dropped:3 overruns:0 carrier:0      collisions:0 txqueuelen:1000      RX
bytes:1596 (1.5 Kb) TX bytes:252 (252.0 b)
```

- GigE interface

```
[user@opte10 user]$ /sbin/ifconfig eth0 eth0 Link encap:Ethernet HWaddr
00:00:1A:19:30:80       inet addr:172.20.0.10 Bcast:172.20.255.255 Mask:
255.0.0.0       UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1       RX
packets:133469120 errors:0 dropped:0 overruns:0  frame:0       TX packets:
135950325 errors:0 dropped:0 overruns:0  carrier:0       collisions:0
txqueuelen:1000      RX bytes:24498382931 (23363.4 Mb) TX bytes:29823673137
 (28442.0Mb)       Interrupt:31
```

Running applications on Windows

Building and running multihost on Windows HPCS clusters

The following is an example of basic compilation and run steps to execute `hello_world.c` on a cluster with 16-way pllelism. To build and run `hello_world.c` on a HPCS cluster:

1. Change to a writable directory on a mapped drive. Share the mapped drive to a folder for the cluster.
2. Open a Visual Studio command window. (This example uses a 64-bit version, so a Visual Studio 64-bit command window is opened.)
3. Compile the hello_world executable file:

   `X:\demo>` **set MPI_CC=cl**

   `X:\demo>` **"%MPI_ROOT%\bin\mpicc" -mpi64 "%MPI_ROOT%\help\hello_world.c"**

   ```
   Microsoft® C/C++ Optimizing Compiler Version 14.00.50727.42 for 64-bit
      Copyright© Microsoft Corporation. All rights reserved.

      hello_world.c
      Microsoft® Incremental Linker Version 8.00.50727.42
      Copyright© Microsoft Corporation. All rights reserved.

      /out:hello_world.exe
      "/libpath:C:\Program Files (x86)\Platform Computing\Platform-MPI\lib"
     /subsystem:console
      libpcmpi64.lib
      libmpio64.lib
      hello_world.obj
   ```
4. Create a new job requesting the number of CPUs to use. Resources are not yet allocated, but the job is given a JOBID number which is printed to stdout:

   `C:\>` **job new /numprocessors:16 /exclusive:true**

   ```
   Job queued, ID: 4288
   ```
5. Add a single-CPU `mpirun` task to the newly created job. The `mpirun` job creates more tasks filling the rest of the resources with the compute ranks, resulting in a total of 16 compute ranks for this example:

   `C:\>` **job add 4288 /numprocessors:1 /exclusive:true /stdout:***node**path**to**a**shared***file.out /stderr:***node**path**to**a**shared***file.err "%MPI_ROOT%\bin\mpirun" -hpc ***node**path**to***hello_world.exe**
6. Submit the job.

   The machine resources are allocated and the job is run.

   `C:\>` **job submit /id:4288**

Run multiple-program multiple-data (MPMD) applications

To run Multiple-Program Multiple-Data (MPMD) applications or other more complex configurations that require further control over the application layout or environment, dynamically create an appfile within the job using the utility `"%MPI_ROOT%\bin\mpi_nodes.exe"` as in the following example. The

environment variable *%CCP_NODES%* cannot be used for this purpose because it only contains the single CPU resource used for the task that executes the `mpirun` command. To create the executable, perform Steps 1 through 3 from the previous section. Then continue with:

1. Create a new job.

   `C:\>` **job new /numprocessors:16 /exclusive:true**

   ```
   Job queued, ID: 4288
   ```

2. Submit a script. Verify *MPI_ROOT* is set in the environment.

   `C:\>` **job add 4288 /numprocessors:1 /env:MPI_ROOT="%MPI_ROOT%" /exclusive:true / stdout:***node**path**to**a**shared***file.out /stderr:***node**path**to**a**shared***file.err** *path* **submission_script.vbs**

   Where `submission_script.vbs` contains code such as:

```
Option Explicit
   Dim sh, oJob, JobNewOut, appfile, Rsrc, I, fs
   Set sh = WScript.CreateObject("WScript.Shell")
   Set fs = CreateObject("Scripting.FileSystemObject")
   Set oJob = sh.exec("%MPI_ROOT%\bin\mpi_nodes.exe")
   JobNewOut = oJob.StdOut.Readall

   Set appfile = fs.CreateTextFile("<path>\appfile", True)

   Rsrc = Split(JobNewOut, " ")

   For I = LBound(Rsrc) + 1 to UBound(Rsrc) Step 2
           appfile.WriteLine("-h" + Rsrc(I) + "-np" + Rsrc(I+1) + _
                   " ""<path>\foo.exe"" ")
   Next

   appfile.Close

   Set oJob = sh.exec("""%MPI_ROOT%\bin\mpirun.exe"" -TCP -f _
             ""<path>\appfile"" ")

   wscript.Echo oJob.StdOut.Readall
```

3. Submit the job as in the previous example:`C:\>` **job submit /id:4288**

The above example using `submission_script.vbs` is only an example. Other scripting languages can be used to convert the output of`mpi_nodes.exe` into an appropriate appfile.

Building an MPI application with Visual Studio and using the property pages

To build an MPI application in C or C++ with VS2008, use the property pages provided by Platform MPI to help link applications.

Two pages are included with Platform MPI, and are located at the installation location in `%MPI_ROOT%\help\HPMPI.vsprops` and `%MPI_ROOT%\help\HPMPI64.vsprops`.

1. Go to VS Project > View > Property Manager and expand the project.

   This displays the different configurations and platforms set up for builds. Include the appropriate property page (`HPMPI.vsprops` for 32-bit applications, `HPMPI64.vsprops` for 64-bit applications) in Configuration > Platform.

2. Select this page by either double-clicking the page or by right-clicking on the page and selecting Properties. Go to the `User Macros` section. Set *MPI_ROOT* to the desired location (for example, the installation location of Platform MPI). This should be set to the default installation location:

`%ProgramFiles(x86)%\Platform Computing\Platform-MPI`.

Note:

This is the default location on 64-bit machines. The location for 32-bit machines is `%ProgramFiles%\Platform Computing\Platform-MPI`

3. The MPI application can now be built with Platform MPI.

   The property page sets the following fields automatically, but can also be set manually if the property page provided is not used:

   1. C/C++ — Additional Include Directories

      Set to "`%MPI_ROOT%\include\[32|64]`"
   2. Linker — Additional Dependencies

      Set to `libpcmpi32.lib` or `libpcmpi64.lib` depending on the application.
   3. Additional Library Directories

      Set to "`%MPI_ROOT%\lib`"

Building and running on a Windows 2008 cluster using appfiles

The example teaches you the basic compilation and run steps to execute `hello_world.c` on a cluster with 4-way pllelism. To build and run `hello_world.c` on a cluster using an appfile, Perform Steps 1 and 2 from Building and Running on a Single Host.

Note:

Specify the bitness using -mpi64 or -mpi32 for `mpicc` to link in the correct libraries. Verify you are in the correct bitness compiler window. Using -mpi64 in a Visual Studio 32-bit command window does not work.

1. Create a file "appfile" for running on nodes n01 and n02 as:

   `C:\>` **-h n01 -np 2 ***node01\share\path\to***hello_world.exe -h n02 -np 2 ***node01\share\path\to***hello_world.exe**
2. For the first run of the hello_world executable, use -cache to cache your password:

   `C:\>` **"%MPI_ROOT%\bin\mpirun" -cache -f appfile**

   Password for MPI runs:

   When typing, the password is not echoed to the screen.

   The Platform MPI Remote Launch service must be registered and started on the remote nodes. `mpirun` will authenticated with the service and create processes using your encrypted password to obtain network resources.

   If you do not provide a password, the password is incorrect, or you use -nopass, remote processes are created but do not have access to network shares. In the following example, the `hello_world.exe` file cannot be read.
3. Analyze hello_world output.

Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
Hello world! I'm 1 of 4 on n01
Hello world! I'm 3 of 4 on n02
Hello world! I'm 0 of 4 on n01
Hello world! I'm 2 of 4 on n02
```

Running with an appfile using HPCS

Using an appfile with HPCS has been greatly simplified in this release of Platform MPI. The previous method of writing a submission script that uses `mpi_nodes.exe` to dynamically generate an appfile based on the HPCS allocation is still supported. However, the preferred method is to allow `mpirun.exe` to determine which nodes are required for the job (by reading the user-supplied appfile), request those nodes from the HPCS scheduler, then submit the job to HPCS when the requested nodes have been allocated. The user writes a brief appfile calling out the exact nodes and rank counts needed for the job. For example:

Perform Steps 1 and 2 from Building and Running on a Single Host.

1. Create an appfile for running on nodes n01 and n02 as:

   -h n01 -np 2 hello_world.exe

   -h n02 -np 2 hello_world.exe

2. Submit the job to HPCS with the following command:

   `X:\demo>` **mpirun -hpc -f appfile**

3. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output.

   ```
   Hello world! I'm 2 of 4 on n02
   Hello world! I'm 1 of 4 on n01
   Hello world! I'm 0 of 4 on n01
   Hello world! I'm 3 of 4 on n02
   ```

Building and running on a Windows 2008 cluster using -hostlist

Perform Steps 1 and 2 from the previous section Building and Running on a Single Host.

1. Run the -cache password if this is your first run of Platform MPI on the node and in this user account.

   Use the -hostlist flag to indicate which hosts to run:

   `X:\demo>` **"%MPI_ROOT%\bin\mpirun" -cache -hostlist n01:2,n02:2 hello_world.exe**

   ```
   Password for MPI runs:
   ```

   This example uses the -hostlist flag to indicate which nodes to run on. Also note that the *MPI_WORKDIR* is set to your current directory. If this is not a network mapped drive, Platform MPI is unable to convert this to a Universal Naming Convention (UNC) path, and you must specify the full UNC path for `hello_world.exe`.

2. Analyze hello_world output.

Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
Hello world! I'm 1 of 4 on n01
Hello world! I'm 3 of 4 on n02
Hello world! I'm 0 of 4 on n01
Hello world! I'm 2 of 4 on n02
```

3. Any future Platform MPI runs can now use the cached password.

   Any additional runs of ANY Platform MPI application from the same node and same user account will not require a password:

   ```
   X:\demo> "%MPI_ROOT%\bin\mpirun" -hostlist n01:2,n02:2 hello_world.exe
   ```

   ```
   Hello world! I'm 1 of 4 on n01
   Hello world! I'm 3 of 4 on n02
   Hello world! I'm 0 of 4 on n01
   Hello world! I'm 2 of 4 on n02
   ```

Running with a hostfile using HPCS

1. Perform Steps 1 and 2 from Building and Running on a Single Host.
2. Change to a writable directory on a mapped drive. The mapped drive must be to a shared folder for the cluster.
3. Create a file "hostfile" containing the list of nodes on which to run:

   ```
   n01
   n02
   n03
   n04
   ```

4. Submit the job to HPCS.

   ```
   X:\demo> "%MPI_ROOT%\bin\mpirun" -hpc -hostfile hfname -np 8 hello_world.exe
   ```

   Nodes are allocated in the order that they appear in the hostfile. Nodes are scheduled cyclically, so if you have requested more ranks than there are nodes in the hostfile, nodes are used multiple times.

5. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

   ```
   Hello world! I'm 5 of 8 on n02
   Hello world! I'm 0 of 8 on n01
   Hello world! I'm 2 of 8 on n03
   Hello world! I'm 6 of 8 on n03
   Hello world! I'm 1 of 8 on n02
   Hello world! I'm 3 of 8 on n04
   Hello world! I'm 4 of 8 on n01
   Hello world! I'm 7 of 8 on n04
   ```

Running with a hostlist using HPCS

Perform Steps 1 and 2 from Building and Running on a Single Host.

1. Change to a writable directory on a mapped drive. The mapped drive should be to a shared folder for the cluster.
2. Submit the job to HPCS, including the list of nodes on the command line.

   ```
   X:\demo> "%MPI_ROOT%\bin\mpirun" -hpc -hostlist n01,n02,n03,n04 -np 8 hello_world.exe
   ```

Nodes are allocated in the order that they appear in the hostlist. Nodes are scheduled cyclically, so if you have requested more ranks than there are nodes in the hostlist, nodes are used multiple times.

3. Analyze hello_world output.

   Platform MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
Hello world! I'm 5 of 8 on n02
Hello world! I'm 0 of 8 on n01
Hello world! I'm 2 of 8 on n03
Hello world! I'm 6 of 8 on n03
Hello world! I'm 1 of 8 on n02
Hello world! I'm 3 of 8 on n04
Hello world! I'm 4 of 8 on n01
Hello world! I'm 7 of 8 on n04
```

Performing multi-HPC runs with the same resources

In some instances, such as when running performance benchmarks, it is necessary to perform multiple application runs using the same set of HPC nodes. The following example is one method of accomplishing this.

1. Compile the `hello_world` executable file.
   a) Change to a writable directory, and copy `hello_world.c` from the help directory:

      `C:\>` **copy "%MPI_ROOT%\help\hello_world.c" .**

   b) Compile the hello_world executable file.

      In a proper compiler command window (for example, Visual Studio command window), use `mpicc` to compile your program:

      `C:\>` **"%MPI_ROOT%\bin\mpicc" -mpi64 hello_world.c**

      Note:

      Specify the bitness using -mpi64 or -mpi32 for `mpicc` to link in the correct libraries. Verify you are in the correct 'bitness' compiler window. Using -mpi64 in a Visual Studio 32-bit command window does not work.

2. Request a HPC allocation of sufficient size to run the required application(s). Add the /rununtilcanceled option to have HPC maintain the allocation until it is explicitly canceled.

   > **job new /numcores:8 /rununtilcanceled:true**

   ```
   Job queued, ID: 4288
   ```

3. Submit the job to HPC without adding tasks.

   > **job submit /id:4288**

   ```
   Job 4288 has been submitted.
   ```

4. Run the applications as a task in the allocation, optionally waiting for each to finish before starting the following one.

   > **"%MPI_ROOT%\bin\mpirun" -hpc -hpcwait -jobid 4288 \\node\share\hello_world.exe**

   ```
   mpirun: Submitting job to hpc scheduler on this node
   mpirun: HPMPI Job 4288 submitted to cluster mpiccp1
   mpirun: Waiting for HPMPI Job 4288 to finish...
   mpirun: HPMPI Job 4288 finished.
   ```

Note:

Platform MPI automatic job submittal converts the mapped drive to a UNC path, which is necessary for the compute nodes to access files correctly. Because this example uses HPCS commands for submitting the job, the user must explicitly indicate a UNC path for the MPI application (i.e., `hello_world.exe`) or include the /workdir flag to set the shared directory as the working directory.

5. Repeat Step 4 until all required runs are complete.
6. Explicitly cancel the job, freeing the allocated nodes.

   > **job cancel 4288**

Remote launch service for Windows

Remote Launch service is available for Windows 2003/XP/Vista/2008/Windows 7 system

The Platform MPI Remote Launch service is located in `"%MPI_ROOT%\sbin\PCMPIWin32Service.exe"`. *MPI_ROOT* must be located on a local disk or the service does not run properly.

To run the service manually, you must register and start the service. To register the service manually, run the service executable with the -i option. To start the service manually, run the service after it is installed with the -start option. The service executable is located at `"%MPI_ROOT%\sbin\PCMPIWin32Service.exe"`.

For example:

```
C:\> "%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -i
Creating Event Log Key
'PCMPI'...Installing service 'Platform-MPI SMPID'...
OpenSCManager OK
CreateService Succeeded
Service installed.
```

```
C:\> "%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -start
Service started...
```

The Platform MPI Remote Launch service runs continually as a Windows service, listening on a port for Platform MPI requests from remote `mpirun.exe` jobs. This port must be the same port on all machines, and is established when the service starts. The default TCP port is 8636.

If this port is not available or to change the port, include a port number as a parameter to -i. As an example, to install the service with port number 5004:

C:\> **"%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -i 5004**

Or, you can stop the service, then set the port key, and start the service again. For example, using port 5004:

C:\> **"%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -stop**

```
Service stopped...
```

C:\> **"%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -setportkey 5004**

```
Setting Default Port key...'PCMPI'...

Port Key set to 5004
```

C:\> **"%MPI_ROOT%\sbin\PCMPIWin32Service.exe" -start**

```
Service started...
```

For additional Platform MPI Remote Launch service options, use -help.

Usage: **pcmpiwin32service.exe [***cmd* **[***pm***]]**

where **cmd** can be one of the following commands:

-? | -h | -help

show command usage

-s | -status

show service status

-k | -removeeventkey

remove service event log key

-r | -removeportkey

remove default port key

-t | -setportkey <port>

remove default port key

-i | -install [<port>]

remove default port key

-start

start an installed service

-stop

stop an installed service

-restart

restart an installed service

Note:

All remote services must use the same port. If you are not using the default port, make sure you select a port that is available on all remote nodes.

Run-time utility commands

Platform MPI provides a set of utility commands to supplement MPI library routines.

mpidiag tool for Windows 2003/XP and Platform MPI Remote Launch Service

Platform MPI for Windows 2003/XP includes the `mpidiag` diagnostic tool. It is located in `%MPI_ROOT%\bin\mpidaig.exe`.

This tool is useful to diagnose remote service access without running `mpirun`. To use the tool, run `mpidiag` with -s <*remote-node*> <*options*>, where options include:

-help

Show the options to `mpidiag`.

-s <*remote-node*>

Connect to and diagnose this node's remote service.

-at

Authenticates with the remote service and returns the remote authenticated user's name.

-st

Authenticates with remote service and returns service status.

-et <*echo-string*>

Authenticates with the remote service and performs a simple echo test, returning the string.

-sys

Authenticates with the remote service and returns remote system information, including node name, CPU count, and username.

-ps [*username*]

Authenticates with the remote service, and lists processes running on the remote system. If a username is included, only that user's processes are listed.

-dir <*path*>

Authenticates with the remote service and lists the files for the given path. This is a useful tool to determine if access to network shares are available to the authenticated user.

-sdir <*path*>

Same as -dir, but lists a single file. No directory contents are listed. Only the directory is listed if accessible.

-kill <*pid*>

Authenticates with remote service and terminates the remote process indicated by the pid. The process is terminated as the authenticated user. If the user does not have permission to terminate the indicated process, the process is not terminated.

`mpidiag` authentication options are the same as `mpirun` authentication options. These include: -pass, -cache, -clearcache, -iscached, -token/-tg, -package/-pk. For detailed descriptions of these options, refer to these options in the `mpirun` documentation.

The `mpidiag` tool can be very helpful in debugging issues with remote launch and access to remote systems through the Platform MPI Remote Launch service. To use the tool, you must always supply a 'server' with the -s option. Then you can use various commands to test access to the remote service, and verify a limited number of remote machine resources.

For example, to test if machine 'winbl16' Platform MPI remote launch service is running, use the -at flag:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -at

connect() failed: 10061

Cannot establish connection with server.
```

```
SendCmd(): send() sent a different number of bytes than expected: 10057
```

The machine cannot connect to the service on the remote machine. After checking (and finding the service was not started), the service is restarted and the command is run again:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -at

Message received from Service: user1
```

Now the service responds and authenticates correctly.

To verify what processes are running on a remote machine, use the -ps command:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -ps

Process List:
ProcessName     Username   PID   CPU Time   Memory
rdpclip.exe     user1      2952  0.046875   5488
explorer.exe    user1      1468  1.640625   17532
reader_sl.exe   user1      2856  0.078125   3912
cmd.exe         user1      516   0.031250   2112
ccApp.exe       user1      2912  0.187500   7580
mpid.exe        user1      3048  0.125000   5828
Pallas.exe      user1      604   0.421875   13308
CMD Finished successfully.
```

The processes by the current user 'user1' runs on 'winbl16'. Two of the processes are MPI jobs: `mpid.exe` and `Pallas.exe`. If these are not supposed to be running, use `mpidiag` to kill the remote process:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -kill 604

CMD Finished successfully.

X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -ps

Process List:
ProcessName     Username   PID   CPU Time   Memory
rdpclip.exe     user1      2952  0.046875   5488
explorer.exe    user1      1468  1.640625   17532
reader_sl.exe   user1      2856  0.078125   3912
cmd.exe         user1      516   0.031250   2112
ccApp.exe       user1      2912  0.187500   7580
CMD Finished successfully.
```

`Pallas.exe` was killed, and Platform MPI cleaned up the remaining Platform MPI processes.

Another useful command is a short 'system info' command, indicating the machine name, system directories, CPU count and memory:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -sys

SystemInfo:
Computer name       : WINBL16
User name           : user1
System Directory    : C:\WINDOWS\system32
Windows Directory   : C:\WINDOWS
CPUs                : 2
TotalMemory         : 2146869248
Small selection of Environment Variables:     OS   = Windows_NT     PATH = C:\Perl\bin
\;C:\WINDOWS\system32;           C:\WINDOWS;C:\WINDOWS\System32\Wbem     HOMEPATH   =
%HOMEPATH%     TEMP = C:\WINDOWS\TEMP
CMD Finished successfully.
```

You can view directories accessible from the remote machine when authenticated by the user:

```
X:\Demo>  "%MPI_ROOT%\bin\mpidiag" -s winbl16 -dir \\mpiccp1\scratch\user1

Directory/File list:
Searching for path: \\mpiccp1\scratch\user1
```

```
Directory: \\mpiccp1\scratch\user1
..
BaseRel
Beta-HPMPI
BuildTests
DDR2-Testing
dir.pl
exportedpath.reg
FileList.txt
h1.xml
HelloWorld-HP64-2960.1.err
HelloWorld-HP64-2960.1.out
HelloWorld-HP64-2961.1.err
HelloWorld-HP64-2961.1.out
```

mpidiag tool for Windows 2008 and Platform MPI Remote Launch Service

Platform MPI for Windows 2008 includes the `mpidiag` diagnostic tool.

It is located in `%MPI_ROOT%\bin\mpidaig.exe`.

This tool is useful to diagnose remote service access without running `mpirun`. To use the tool, run `mpidiag` with -s <*remote-node*> <*options*>, where options include:

-help

Show the options to `mpidiag`.

-s <*remote-node*>

Connect to and diagnose the remote service of this node.

-at

Authenticates with the remote service and returns the remote authenticated user's name.

-st

Authenticates with remote service and returns service status.

-et <*echo-string*>

Authenticates with the remote service and performs a simple echo test, returning the string.

-sys

Authenticates with the remote service and returns remote system information, including node name, CPU count, and username.

-ps [*username*]

Authenticates with the remote service and lists processes running on the remote system. If a username is included, only that user's processes are listed.

-dir <*path*>

Authenticates with the remote service and lists the files for the given path. This is a useful tool to determine if access to network shares are available to the authenticated user.

-sdir <*path*>

Same as -dir, but lists a single file. No directory contents are listed. Only the directory is listed if accessible.

-kill *<pid>*

Authenticates with remote service and terminates the remote process indicated by the pid. The process is terminated as the authenticated user. So, if the user does not have permission to terminate the indicated process, the process will not be terminated.

Note:

`mpidiag` authentication options are the same as `mpirun` authentication options. These include: -pass, -cache, -clearcache, -iscached, -token/-tg, -package/-pk.

mpiexec

The MPI-2 standard defines `mpiexec` as a simple method to start MPI applications. It supports fewer features than `mpirun`, but it is portable. `mpiexec` syntax has three formats:

- `mpiexec` offers arguments similar to a MPI_Comm_spawn call, with arguments as shown in the following form:

  `mpiexec` *`mpiexec-options command command-args`*

  For example:

  `C:\>` **%MPI_ROOT%\bin\mpiexec /cores 8 myprog.x 1 2 3**

  creates an 8 rank MPI job on the local host consisting of 8 copies of the program myprog.x, each with the command-line arguments 1, 2, and 3.
- It also allows arguments like a MPI_Comm_spawn_multiple call, with a colon-separated list of arguments, where each component is like the form above.

  For example:

  `C:\>` **%MPI_ROOT%\bin\mpiexec /cores 4 myprog.x : /host host2 /cores 4 \path\to\myprog.x**

  creates a MPI job with 4 ranks on the local host and 4 on host 2.
- Finally, the third form allows the user to specify a file containing lines of data like the arguments in the first form.

  `mpiexec [/configfile` *`file`*`]`

  For example:

  `C:\>` **%MPI_ROOT%\bin\mpiexec /configfile cfile**

  gives the same results as in the second example, but using the /configfile option (assuming the `cfile` file contains `/cores 4 myprog.x /host host2 /cores 4 /wdir /some/path myprog.x`)

The following `mpiexec` options are those whose contexts affect the whole command line:

/cores *number*

Ranks-per-host to use if not specified elsewhere. This applies when processing the `/ghosts`, `/gmachinefile`, `/hosts`, and `/machinefile` options.

/affinity

Enables Platform MPI's `-cpu_bind` option.

/gpath *path*[;*path1* ...]

Prepends file paths to the *PATH* environment variable.

/lines

Enables Platform MPI's `-stdio=p` option.

/genv *variable value* or -genv *variable value*

Uses Platform MPI's `-e` *variable=value* option.

/genvlist *var1*[,*var2* ...]

This option is similar to `/genv`, but uses `mpirun`'s current environment for the variable values.

/gdir *directory* or -dir *directory*

Uses Platform MPI's `-e MPI_WORKDIR=`*directory* option.

/gwdir *directory* or -wdir *directory*

Uses Platform MPI's `-e MPI_WORKDIR=`*directory* option.

/ghost *host_name*

Each portion of the command line where a host (or hosts) are not explicitly specified is run under the "default context". `/ghost` *host_name* sets this default context to *host_name* with np=1.

/ghosts *num hostA* numA *hostB numB* ...

This option is similar to `/ghost`, but sets the default context to the specified list of hosts and np settings. Unspecified np settings are either 1 or whatever was specified in `/cores` *number*, if used.

/gmachinefile *file*

This option is similar to `/ghosts`, but the *hostx numx* settings are read from the specified file.

The following options are those whose contexts only affect the current portion of the command line:

/np *number*

Specifies the number of ranks to launch onto whatever hosts are represented by the current context.

/host *host_name*

Sets the current context to host_name with np=1.

/hosts *num hostA* numA *hostB numB* ...

This option is similar to `/ghosts`, and sets the current context.

/machinefile *file*

This option is similar to `/hosts`, but the *hostx numx* settings are read from the specified file.

/wdir *dir*

The local-context version of `/gdir`.

/env *variable value*

The local-context version of `/genv`.

/envlist *var1*[,*var2* ...]

The local-context version of `/genvlist`.

/path *path*[;*path1* ...]

The local-context version of `/gpath`.

The following are additional options for MPI:

/quiet_hpmpi

By default, Platform MPI displays a detailed account of the types of MPI commands that are translated to assist in determining if the result is correct. This command disables these messages.

`mpiexec` does not support `prun` or `srun` start-up.

mpirun options

This section describes options included in <*mpirun_options*> for all of the preceding examples. They are listed by category:

- Interconnect selection
- Launching specifications
- Debugging and informational
- RDMA control
- MPI-2 functionality
- Environment control
- Special Platform MPI mode
- Windows CCP
- Windows 2003/XP

Interconnect selection options

Network selection

-ibv/-IBV

Explicit command-line interconnect selection to use OFED InfiniBand. The lowercase and uppercase options are analogous to the Elan options.

-vapi/-VAPI

Explicit command-line interconnect selection to use Mellanox Verbs API. The lowercase and uppercase options are analogous to the Elan options.

Dynamic linking is required with VAPI. Do not link -static.

-udapl/-UDAPL

Explicit command-line interconnect selection to use uDAPL. The lowercase and uppercase options are analogous to the Elan options.

Dynamic linking is required with uDAPL. Do not link -static.

-psm/-PSM

Explicit command-line interconnect selection to use QLogic InfiniBand. The lowercase and uppercase options are analogous to the Elan options.

-mx/-MX

Explicit command-line interconnect selection to use Myrinet MX. The lowercase and uppercase options are analogous to the Elan options.

-gm/-GM

Explicit command-line interconnect selection to use Myrinet GM. The lowercase and uppercase options are analogous to the Elan options.

-elan/-ELAN

Explicit command-line interconnect selection to use Quadrics Elan. The lowercase option is taken as advisory and indicates that the interconnect should be used if it is available. The uppercase option is taken as mandatory and instructs MPI to abort if the interconnect is unavailable. The interaction between these options and the related *MPI_IC_ORDER* variable is that any command line interconnect selection here is implicitly prepended to *MPI_IC_ORDER*.

-itapi/-ITAPI

Explicit command-line interconnect selection to use ITAPI. The lowercase and uppercase options are analogous to the Elan options.

-ibal/-IBAL

Explicit command-line interconnect selection to use the Windows IB Access Layer. The lowercase and uppercase options are analogous to the Elan options.

Platform MPI for Windows supports automatic interconnect selection. If a valid InfiniBand network is found, IBAL is selected automatically. It is no longer necessary to explicitly specify `-ibal/-IBAL`.

-TCP

Specifies that TCP/IP should be used instead of another high-speed interconnect. If you have multiple TCP/IP interconnects, use -netaddr to specify which interconnect to use. Use -prot to see which interconnect was selected. Example:

```
% $MPI_ROOT/bin/mpirun -TCP -hostlist "host1:4,host2:4" -np8 ./a.out
```

-commd

Routes all off-host communication through daemons rather than between processes. (Not recommended for high-performance solutions.)

Local host communication method

-intra=mix

Use shared memory for small messages. The default is 256 KB, or what is set by *MPI_RDMA_INTRALEN*. For larger messages, the interconnect is used for better bandwidth. This same functionality is available through the environment variable *MPI_INTRA* which can be set to shm, nic, or mix.

This option does not work with TCP, Elan, MX, or PSM.

-intra=nic

Use the interconnect for all intrahost data transfers. (Not recommended for high performance solutions.)

-intra=shm

Use shared memory for all intrahost data transfers. This is the default.

TCP interface selection

-netaddr

This option is similar to -subnet, but allows finer control of the selection process for TCP/IP connections. MPI has two main sets of connections: those between ranks and/or daemons where real message traffic occurs, and connections between `mpirun` and the daemons where little traffic occurs (but is still necessary).

The -netaddr option can be used to specify a single IP/mask to use for both purposes, or specify them individually. The latter might be needed if `mpirun` happens to be run on a remote machine that doesn't have access to the same Ethernet network as the rest of the cluster. To specify both, the syntax is -netaddr *IP-specification*[/*mask*]. To specify them individually, the syntax is -netaddr mpirun:*spec*,*rank*:*spec*. The string `launch:` can be used in place of `mpirun:`.

The IP specification can be a numeric IP address like 172.20.0.1 or it can be a host name. If a host name is used, the value is the first IP address returned by gethostbyname(). The optional mask can be specified as a dotted quad, or as a number representing how many bits are to be matched. For example, a mask of "11" is equivalent to a mask of "255.224.0.0".

If an IP and mask are given, then it is expected that one and only one IP will match at each lookup. An error or warning is printed as appropriate if there are no matches, or too many. If no mask is specified, then the IP matching will simply be done by the longest matching prefix.

This functionality can also be accessed using the environment variable *MPI_NETADDR*.

Launching specifications options

Job launcher/scheduler

Options for LSF users

These options launch ranks as found in appfile mode on the hosts specified in the environment variable.

-lsb_hosts

Launches the same executable across multiple hosts. Uses the list of hosts in the environment variable *$LSB_HOSTS*. Can be used with the -np option.

-lsb_mcpu_hosts

Launches the same executable across multiple hosts. Uses the list of hosts in the environment variable *$LSB_MCPU_HOSTS*. Can be used with the -np option.

-lsf

Launches the same executable across multiple hosts. Uses the list of hosts in the environment variable *$LSB_MCPU_HOSTS* and sets MPI_REMSH to use LSF's ssh replacement, `blaunch` .

Note:

blaunch requires LSF 7.0.6 and up.

Platform MPI integrates features for jobs scheduled and launched through Platform LSF. These features require Platform LSF 7.0.6 or later. Platform LSF 7.0.6 introduced the `blaunch` command as an ssh-like remote shell for launching jobs on nodes allocated by LSF. Using `blaunch` to start remote processes allows for better job accounting and job monitoring through LSF. When submitting an mpirun job to LSF `bsub`, either add the `-lsf mpirun` command line option or set the variable -e MPI_USELSF=y in the job submission environment. These two options are equivalent. Setting either of the options automatically sets both the `-lsb_mcpu_hosts mpirun` command line option and the MPI_REMSH=blaunch environment variable in the mpirun environment when the job is executed.

Example:

bsub -I -n 4 $MPI_ROOT/bin/mpirun -TCP -netaddr 123.456.0.0 -e MPI_ROOT=$MPI_ROOT -lsf ./hello_world

```
Job <189> is submitted to default queue <normal>.
<<Waiting for dispatch ...>>
<<Starting on example.platform.com>>
Hello world! I'm 0 of 4 on n01
Hello world! I'm 2 of 4 on n01
Hello world! I'm 1 of 4 on n01
Hello world! I'm 3 of 4 on n01
```

Options for `prun` users

-prun

Enables start-up with Elan usage. Only supported when linking with shared libraries. Some features like `mpirun -stdio` processing are unavailable. The -np option is not allowed with -prun. Arguments on the `mpirun` command line that follow -prun are passed to the `prun` command.

Options for SLURM users

-srun

Enables start-up on HP XC clusters. Some features like `mpirun -stdio` processing are unavailable. The -np option is not allowed with -srun. Arguments on the `mpirun` command line that follow -srun are passed to the `srun` command. Start-up directly from the `srun` command is not supported.

Remote shell launching

-f *appfile*

Specifies the appfile that `mpirun` parses to get program and process count information for the run.

-hostfile *<filename>*

Launches the same executable across multiple hosts. File name is a text file with host names separated by spaces or new lines. Can be used with the -np option.

-hostlist *<list>*

Launches the same executable across multiple hosts. Can be used with the -np option. This host list can be delimited with spaces or commas. Hosts can be followed with an optional rank count, which is delimited from the host name with a space or colon. If spaces are used as delimiters in the host list, it might be necessary to place the entire host list inside quotes to prevent the command shell from interpreting it as multiple options.

-h*host*

Specifies a host on which to start the processes (default is *local_host*). Only applicable when running in single host mode (`mpirun -np ...`). See the -hostlist option which provides more flexibility.

-np #

Specifies the number of processes to run. Generally used in single host mode, but also valid with -hostfile, -hostlist, -lsb_hosts, and -lsb_mcpu_hosts.

-stdio=[*options*]

Specifies standard IO options. This applies to appfiles only.

Process placement

-cpu_bind

Binds a rank to an ldom to prevent a process from moving to a different ldom after start-up.

Application bitness specification

-mpi32

Option for running on Opteron and Intel64. Should be used to indicate the bitness of the application to be invoked so that the availability of interconnect libraries can be properly determined by the Platform MPI utilities `mpirun` and `mpid`. The default is -mpi64.

-mpi64

Option for running on Opteron and Intel64. Should be used to indicate the bitness of the application to be invoked so that the availability of interconnect libraries can be properly determined by the Platform MPI utilities `mpirun` and `mpid`. The default is -mpi64.

Debugging and informational options

-help

Prints usage information for `mpirun`.

-version

Prints the major and minor version numbers.

-prot

Prints the communication protocol between each host (e.g., TCP/IP or shared memory). The exact format and content presented by this option is subject to change as new interconnects and communication protocols are added to Platform MPI.

-ck

Behaves like the -p option, but supports two additional checks of your MPI application; it checks if the specified host machines and programs are available, and also checks for access or permission problems. This option is only supported when using appfile mode.

-d

Debug mode. Prints additional information about application launch.

-j

Prints the Platform MPI job ID.

-p

Turns on pretend mode. The system starts a Platform MPI application but does not create processes. This is useful for debugging and checking whether the appfile is set up correctly. This option is for appfiles only.

-v

Turns on verbose mode.

-i *spec*

Enables run time instrumentation profiling for all processes. spec specifies options used when profiling. The options are the same as those for the environment variable *MPI_INSTR*. For example, the following is valid:

```
% $MPI_ROOT/bin/mpirun -i mytrace:l:nc -f appfile
```

Lightweight instrumentation can be turned on by using either the -i option to `mpirun` or by setting the environment variable *MPI_INSTR*.

Instrumentation data includes some information on messages sent to other MPI worlds formed using MPI_Comm_accept(), MPI_Comm_connect(), or MPI_Comm_join(). All off-world message data is accounted together using the designation `offw` regardless of which off-world rank was involved in the communication.

Platform MPI provides an API that enables users to access the lightweight instrumentation data on a per-process basis before the application calling MPI_Finalize (). The following declaration in C is necessary to access this functionality:

extern int hpmp_instrument_runtime(int reset)

A call to hpmp_instrument_runtime(0) populates the output file specified by the -i option to `mpirun` or the *MPI_INSTR* environment variable with the statistics available at the time of the call. Subsequent calls to hpmp_instrument_runtime() or MPI_Finalize () will overwrite the contents of the specified file. A call to hpmp_instrument_runtime

(1) populates the file in the same way, but also resets the statistics. If instrumentation is not being used, the call to hpmp_instrument_runtime() has no effect.

For an explanation of -i options, refer to the `mpirun` documentation.

-T

Prints user and system times for each MPI rank.

-dbgspin

Causes each rank of the MPI application to spin in MPI_INIT(), allowing time for the user to log in to each node running the MPI application and attach a debugger to each process. Setting the global variable *mpi_debug_cont* to a non-zero value in the debugger will allow that process to continue. This is similar to the debugging methods described in the mpidebug(1) manpage, except that -dbgspin requires the user to launch and attach the debuggers manually.

-tv

Specifies that the application runs with the TotalView debugger.

RDMA control options

-dd

Uses deferred deregistration when registering and deregistering memory for RDMA message transfers. The default is to use deferred deregistration. Note that using this option also produces a statistical summary of the deferred deregistration activity when *MPI_Finalize* is called. The option is ignored if the underlying interconnect does not use an RDMA transfer mechanism, or if the deferred deregistration is managed directly by the interconnect library.

Occasionally deferred deregistration is incompatible with an application or negatively impacts performance. Use -ndd to disable this feature.

Deferred deregistration of memory on RDMA networks is not supported on Platform MPI for Windows.

-ndd

Disables the use of deferred deregistration. For more information, see the -dd option.

-rdma

Specifies the use of envelope pairs for short message transfer. The prepinned memory increases proportionally to the number of off-host ranks in the job.

-srq

Specifies use of the shared receiving queue protocol when OFED, Myrinet GM, ITAPI, Mellanox VAPI or uDAPL V1.2 interfaces are used. This protocol uses less prepinned memory for short message transfers.

-xrc

Extended Reliable Connection (XRC) is a feature on ConnectX InfiniBand adapters. Depending on the number of application ranks that are allocated to each host, XRC can

significantly reduce the amount of pinned memory that is used by the InfiniBand driver. Without XRC, the memory amount is proportional to the number of ranks in the job. With XRC, the memory amount is proportional to the number of hosts on which the job is being run.

The -xrc option is equivalent to -srq -e MPI_IBV_XRC=1.

OFED version 1.3 or later is required to use XRC.

MPI-2 functionality options

-1sided

Enables one-sided communication. Extends the communication mechanism of Platform MPI by allowing one process to specify all communication parameters, for the sending side and the receiving side.

The best performance is achieved if an RDMA-enabled interconnect, like InfiniBand, is used. With this interconnect, the memory for the one-sided windows can come from *MPI_Alloc_mem* or from malloc. If TCP/IP is used, the performance will be lower, and the memory for the one-sided windows must come from *MPI_Alloc_mem*.

-spawn

Enables dynamic processes.

Environment control options

-e *var* [=*val*]

Sets the environment variable var for the program and gives it the value val if provided. Environment variable substitutions (for example, $FOO) are supported in the val argument. To append settings to a variable, %VAR can be used.

-sp *paths*

Sets the target shell *PATH* environment variable to paths. Search paths are separated by a colon.

Special Platform MPI mode option

-ha

Eliminates an MPI teardown when ranks exit abnormally. Further communications involved with ranks that are unreachable return error class *MPI_ERR_EXITED*, but the communications do not force the application to teardown, if the MPI_Errhandler is set to *MPI_ERRORS_RETURN*.

This mode never uses shared memory for inter-process communication.

Platform MPI high availability mode is accessed by using the -ha option to `mpirun`.

To allow users to select the correct level of high availability features for an application, the -ha option accepts a number of additional colon-separated options which may be appended to the -ha command line option. For example:

mpirun -ha[:*option1***][:***option2***][...]**

Table 16: High availability options

Options	Descriptions
-ha	Basic high availability protection. When specified with no options, -ha is equivalent to -ha:noteardown:detect.
-ha -i	Use of lightweight instrumentation with -ha.
-ha:infra	High availability for infrastructure (`mpirun`, `mpid`).
-ha:detect	Detection of failed communication connections.
-ha:recover	Recovery of communication connections after failures.
-ha:net	Enables Automatic Port Migration.
-ha:noteardown	`mpirun` and `mpid` exist, they should not tear down an application in which some ranks have exited after *MPI_Init*, but before *MPI_Finalize*. If -ha:infra is specified, this option is ignored.
-ha:all	-ha:all is equivalent to -ha:infra:noteardown:recover:detect:net, which is equivalent to -ha:infra:recover:net.

Note:

If a process uses -ha:detect, then all processes it communicates with must also use -ha:detect. Likewise, if a process uses -ha:recover then all processes it communicates with must also use -ha:recover.

Support for high availability on InfiniBand Verbs

You can use the -ha option with the -IBV option. When using -ha, automatic network selection is restricted to TCP and IBV. Be aware that -ha no longer forces the use of TCP.

If TCP is desired on a system that has both TCP and IBV available, it is necessary to explicitly specify -TCP on the `mpirun` command line. All high availability features are available on both TCP and IBV interconnects.

Highly available infrastructure (-ha:infra)

The -ha option allows MPI ranks to be more tolerant of system failures. However, failures can still affect the `mpirun` and `mpid` processes used to support Platform MPI applications.

When the `mpirun`/`mpid` infrastructure is affected by failures, it can affect the application ranks and the services provided to those ranks. Using -ha:infra indicates that the `mpirun` and `mpid` processes normally used to support the application ranks are terminated after all ranks have called MPI_Init().

This option implies -stdio=none. To record stdout and stderr, consider using the -stdio=files option when using -ha:infra.

Because the `mpirun` and `mpid` processes do not persist for the length of the application run, some features are not supported with -ha:infra. These include -spawn, -commd, -1sided.

Using -ha:infra does not allow a convenient way to terminate all ranks associated with the application. It is the responsibility of the user to have a mechanism for application teardown.

Using MPI_Comm_connect and MPI_Comm_accept

MPI_Comm_connect and MPI_Comm_accept can be used without the -spawn option to `mpirun`. This allows applications launched using the -ha:infra option to call these routines. When using high availability mode, these routines do not deadlock even if the remote process exits before, during, or after the call.

Using MPI_Comm_disconnect

In high availability mode, MPI_Comm_disconnect is collective only across the local group of the calling process. This enables a process group to independently break a connection to the remote group in an intercommunicator without synchronizing with those processes. Unreceived messages on the remote side are buffered and might be received until the remote side calls MPI_Comm_disconnect.

Receive calls that cannot be satisfied by a buffered message fail on the remote processes after the local processes have called MPI_Comm_disconnect. Send calls on either side of the intercommunicator fail after either side has called MPI_Comm_disconnect.

Instrumentation and high availability mode

Platform MPI lightweight instrumentation is supported when using -ha and singletons. In the event that some ranks terminate during or before MPI_Finalize(), then the lowest rank id in *MPI_COMM_WORLD* produces the instrumentation output file on behalf of the application and instrumentation data for the exited ranks is not included.

Failure Recover (-ha:recover)

Fault-tolerant MPI_Comm_dup() that excludes failed ranks.

When using -ha:recover, the functionality of MPI_Comm_dup() enables an application to recover from errors.

Important:

The MPI_Comm_dup() function is not standard compliant because a call to MPI_Comm_dup() always terminates all outstanding communications with failures on the communicator regardless of the presence or absence of errors.

When one or more pairs of ranks within a communicator are unable to communicate because a rank has exited or the communication layers have returned errors, a call to *MPI_Comm_dup* attempts to return the largest communicator containing ranks that were fully interconnected at some point during the *MPI_Comm_dup* call. Because new errors can occur at any time, the returned communicator might not be completely error free. However, the two ranks in the original communicator that were unable to communicate before the call are not included in a communicator generated by *MPI_Comm_dup*.

Communication failures can partition ranks into two groups, A and B, so that no rank in group A can communicate to any rank in group B and vice versa. A call to MPI_Comm_dup() can behave similarly to a call to MPI_Comm_split(), returning different legal communicators to different callers. When a larger communicator exists than the largest communicator the rank can join, it returns *MPI_COMM_NULL*.

However, extensive communication failures, such as a failed switch, can make such knowledge unattainable to a rank and result in splitting the communicator.

If the communicator returned by rank A contains rank B, then either the communicator return by ranks A and B will be identical or rank B will return *MPI_COMM_NULL* and any attempt by rank A to communicate with rank B immediately returns *MPI_ERR_EXITED*. Therefore, any legal use of communicator return by MPI_Comm_dup() should not result in a deadlock. Members of the resulting communicator either agree to membership or are unreachable to all members. Any attempt to communicate with unreachable members results in a failure.

Interruptible collectives

When a failure (host, process, or interconnect) that affects a collective operation occurs, at least one rank calling the collective returns with an error. The application must initiate a recovery from those ranks by calling MPI_Comm_dup() on the communicator used by the failed collective. This ensures that all other ranks within the collective also exit the collective. Some ranks might exit successfully from a collective call while other ranks do not. Ranks which exit with *MPI_SUCCESS* will have successfully completed their role in the operation, and any output buffers will be correctly set. The return value of *MPI_SUCCESS* does not indicate that all ranks have successfully completed their role in the operation.

After a failure, one or more ranks must call MPI_Comm_dup(). All future communication on that communicator results in failure for all ranks until each rank has called MPI_Comm_dup() on the communicator. After all ranks have called MPI_Comm_dup(), the parent communicator can be used for point-to-point communication. MPI_Comm_dup() can be called successfully even after a failure. Because the results of a collective call can vary by rank, ensure that an application is written to avoid deadlocks. For example, using multiple communicators can be very difficult as the following code demonstrates:

```
... err = MPI_Bcast(buffer, len, type, root, commA); if (err) {
   MPI_Error_class(err, &class);
if (class == MPI_ERR_EXITED) {
     err = MPI_Comm_dup(commA, &new_commA);
     if (err != MPI_SUCCESS) {
        cleanup_and_exit();
     }
     MPI_Comm_free(commA);
     commA = new_commA;
   } } err = MPI_Sendrecv_replace(buffer2, len2, type2, src, tag1, dest, tag2, commB,
&status); if (err) {
.... ...
```

In this case, some ranks exit successfully from the MPI_Bcast() and move onto the MPI_Sendrecv_replace () operation on a different communicator. The ranks that call MPI_Comm_dup() only cause operations on commA to fail. Some ranks cannot return from the MPI_Sendrecv_replace() call on commB if their partners are also members of commA and are in the call to MPI_Comm_dup() call on commA. This demonstrates the importance of using care when dealing with multiple communicators. In this example, if the intersection of commA and commB is *MPI_COMM_SELF*, it is simpler to write an application that does not deadlock during failure.

Network high availability (-ha:net)

The net option to -ha turns on any network high availability. Network high availability attempts to insulate an application from errors in the network. In this release, -ha:net is only significant on IBV for OFED 1.2 or later, where Automatic Path Migration is used. This option has no effect on TCP connections.

Failure detection (-ha:detect)

When using the -ha:detect option, a communication failure is detected and prevents interference with the application's ability to communicate with other processes that have not been affected by the failure.

In addition to specifying -ha:detect, *MPI_Errhandler* must be set to *MPI_ERRORS_RETURN* using the *MPI_Comm_set_errhandler* function. When an error is detected in a communication, the error class *MPI_ERR_EXITED* is returned for the affected communication. Shared memory is not used for communication between processes.

Only IBV and TCP are supported. This mode cannot be used with the diagnostic library.

Clarification of the functionality of completion routines in high availability mode

Requests that cannot be completed because of network or process failures result in the creation or completion functions returning with the error code *MPI_ERR_EXITED*. When waiting or testing multiple requests using MPI_Testany(), MPI_Testsome(), MPI_Waitany() or MPI_Waitsome(), a request that cannot be completed because of network or process failures is considered a completed request and these routines return with the flag or outcount argument set to non-zero. If some requests completed successfully and some requests completed because of network or process failure, the return value of the routine is *MPI_ERR_IN_STATUS*. The status array elements contain *MPI_ERR_EXITED* for those requests that completed because of network or process failure.

Important:

When waiting on a receive request that uses *MPI_ANY_SOURCE* on an intracommunicator, the request is never considered complete due to rank or interconnect failures because the rank that created the receive request can legally match it. For intercommunicators, after all processes in the remote group are unavailable, the request is considered complete and, the *MPI_ERROR* field of the MPI_Status() structure indicates *MPI_ERR_EXITED*.

MPI_Waitall() waits until all requests are complete, even if an error occurs with some requests. If some requests fail, *MPI_IN_STATUS* is returned. Otherwise, *MPI_SUCCESS* is returned. In the case of an error, the error code is returned in the status array.

Windows HPC

The following are specific `mpirun` command-line options for Windows HPC users.

-hpc

Indicates that the job is being submitted through the Windows HPC job scheduler/launcher. This is the recommended method for launching jobs and is required for all HPC jobs.

-hpcerr <*filename*>

Assigns the job's standard error file to the file name when starting a job through the Windows HPC automatic job scheduler/launcher feature of Platform MPI. This flag has no effect if used for an existing HPC job.

-hpcin <*filename*>

Assigns the job's standard input file to the given file name when starting a job through the Windows HPC automatic job scheduler/launcher feature of Platform MPI. This flag has no effect if used for an existing HPC job.

-hpcout <*filename*>

Assigns the job's standard output file to the given file name when starting a job through the Windows HPC automatic job scheduler/launcher feature of Platform MPI. This flag has no effect if used for an existing HPC job.

-hpcwait

Causes the `mpirun` command to wait for the HPC job to finish before returning to the command prompt when starting a job through automatic job submittal feature of Platform MPI. By default, `mpirun` automatic jobs will not wait. This flag has no effect if used for an existing HPC job.

-hpcblock

Uses block scheduling to place ranks on allocated nodes. Nodes are processed in the order they were allocated by the scheduler, with each node being fully populated up to the total number of CPUs before moving on to the next node. Only valid when the -hpc option is used. Cannot be used with the -f, -hostfile, or -hostlist options.

-hpccluster <*headnode*>

Specifies the headnode of the HPC cluster that should be used to run the job. Assumed to be the local host if omitted.

-hpccyclic

Uses cyclic scheduling to place ranks on allocated nodes. Nodes are processed in the order they were allocated by the scheduler, with one rank allocated per node on each cycle through the node list. The node list is traversed as many times as necessary to reach the total rank count requested. Only valid when the -hpc option is used. Cannot be used with the -f, -hostfile, or -hostlist options.

-headnode <*headnode*>

Indicates the head node to submit the `mpirun` job on Windows HPC. If omitted, local host is used. This option can only be used as a command-line option when using the `mpirun` automatic submittal functionality.

-hosts

Allows you to specify a node list to Platform-MP Ion Windows HPC. Ranks are scheduled according to the host list. The nodes in the list must be in the job allocation or a scheduler error occurs. The Platform MPI program `%MPI_ROOT%\bin \mpi_nodes.exe` returns a string in the proper -hosts format with scheduled job resources.

-jobid <*job-id*>

Schedules a Platform MPI job as a task to an existing job on Windows HPC. It submits the command as a single CPU `mpirun` task to the existing job indicated by the parameter job-id. This option can only be used as a command-line option when using the `mpirun` automatic submittal functionality.

-nodex

In addition to -hpc, indicates that only one rank is to be used per node, regardless of the number of CPUs allocated with each host. This flag is used on Windows HPC.

Windows 2003/XP

The following are specific `mpirun` command-line options for Windows 2003/XP users.

-package <*package-name*>-pk <*package-name*>

When Platform MPI authenticates with the Platform MPI Remote Launch service, it authenticates using an installed Windows security package (e.g., Kerberos, NTLM, Negotiate, etc.). By default, Platform MPI negotiates the package to use with the service, and no interaction or package specification is required by the user. If a specific installed Windows security package is preferred, use this flag to indicate the security package on the client. This flag is rarely necessary because the client (`mpirun`) and the server (Platform MPI Remote Launch service) negotiate the security package to be used for authentication.

-token <*token-name*>-tg <*token-name*>

Authenticates to this token with the Platform MPI Remote Launch service. Some authentication packages require a token name. The default is no token.

-pass

Prompts the user for his domain account password. Used to authenticate and create remote processes. A password is required to allow the remote process to access network resources (such as file shares). The password provided is encrypted using SSPI for authentication. The password is not cached when using this option.

-cache

Prompts the user for the domain account password. Used to authenticate and create remote processes. A password is required to allow the remote process to access network resources (such as file shares). The password provided is encrypted using SSPI for authentication. The password is cached so that future `mpirun` commands use the cached password. Passwords are cached in encrypted form, using Windows Encryption APIs.

-nopass

Executes the `mpirun` command with no password. If a password is cached, it is not accessed and no password is used to create the remote processes. Using no password results in the remote processes not having access to network resources. This option also suppresses the 'no password cached' warning. This is useful when no password is wanted for SMP jobs.

-iscached

Indicates if a password is stored in the user password cache and stops execution. The MPI application will not launch if this option is included on the command line.

-clearcache

Clears the password cache and stops. The MPI application will not launch if this option is included on the command line.

Password authentication

The following are specific `mpirun` command-line options for password authentication.

-pwcheck

Validates the cached user password by obtaining a login token locally and verifying the password. A pass/fail message is returned before exiting.

To check password and authentication on remote nodes, use the -at flag with `mpidiag`.

Note:

The `mpirun` -pwcheck option, along with other Platform MPI password options, run with Platform MPI Remote Launch Service, and do not refer to Windows HPC user passwords. When running through Windows HPC scheduler (with -hpc), you might need to cache a password through the Windows HPC scheduler. For more information, see the Windows HPC job command.

-package <*package-name*> and -pk <*package-name*>

When Platform MPI authenticates with the Platform MPI Remote Launch service, it authenticates using an installed Windows security package (for example Kerberos, NTLM, Negotiate, and more). By default, Platform MPI negotiates the package to use with the service, and no interaction or package specification is required. If a specific installed Windows security package is preferred, use this flag to indicate that security package on the client. This flag is rarely necessary as the client (`mpirun`) and the server (Platform MPI Remote Launch service) negotiates the security package to be used for authentication.

-token <*token-name*> and -tg <*token-name*>

Authenticates to this token with the Platform MPI Remote Launch service. Some authentication packages require a token name. The default is no token.

-pass

Prompts for a domain account password. Used to authenticate and create remote processes. A password is required to allow the remote process to access network resources (such as file shares). The password provided is encrypted using SSPI for authentication. The password is not cached when using this option.

-cache

Prompts for a domain account password. Used to authenticate and create remote processes. A password is required to allow the remote process to access network resources (such as file shares). The password provided is encrypted using SSPI for authentication. The password is cached so that future `mpirun` commands uses the cached password. Passwords are cached in encrypted form, using Windows Encryption APIs.

-nopass

Executes the `mpirun` command with no password. If a password is cached, it is not accessed and no password is used to create the remote processes. Using no password results in the remote processes not having access to network resources. Use this option

if you are running locally. This option also suppresses the "`no password cached`" warning. This is useful when no password is desired for SMP jobs.

-iscached

Indicates if a password is stored in the user password cache and stops execution. The MPI application does not launch if this option is included on the command line.

-clearcache

Clears the password cache and stops. The MPI application does not launch if this option is included on the command line.

Runtime environment variables

Environment variables are used to alter the way Platform MPI executes an application. The variable settings determine how an application behaves and how an application allocates internal resources at run time.

Many applications run without setting environment variables. However, applications that use a large number of nonblocking messaging requests, require debugging support, or must control process placement might need a more customized configuration.

Launching methods influence how environment variables are propagated. To ensure propagating environment variables to remote hosts, specify each variable in an appfile using the-e option.

Setting environment variables on the command line for Linux

Environment variables can be set globally on the `mpirun` command line. Command-line options take precedence over environment variables. For example, on Linux:

% $MPI_ROOT/bin/mpirun -e MPI_FLAGS=y40 -f appfile

In the above example, if an *MPI_FLAGS* setting was specified in the appfile, then the global setting on the command line would override the setting in the appfile. To add to an environment variable rather than replacing it, use %*VAR* as in the following command:

% $MPI_ROOT/bin/mpirun -e MPI_FLAGS=%MPI_FLAGS,y -f appfile

In the above example, if the appfile specified *MPI_FLAGS*=z, then the resulting *MPI_FLAGS* seen by the application would be z, y.

% $MPI_ROOT/bin/mpirun -e LD_LIBRARY_PATH=%LD_LIBRARY_PATH:/path/to/third/party/lib -f appfile

In the above example, the user is appending to *LD_LIBRARY_PATH*.

Setting environment variables in an pcmpi.conf file

Platform MPI supports setting environment variables in an `pcmpi.conf` file. These variables are read by `mpirun` and exported globally, as if they had been included on the `mpirun` command line as "-e VAR=VAL" settings. The `pcmpi.conf` file search is performed in three places and each one is parsed, which allows the last one parsed to overwrite values set by the previous files. The locations are:

- `$MPI_ROOT/etc/pcmpi.conf`
- `/etc/pcmpi.conf`
- `$HOME/.pcmpi.conf`

This feature can be used for any environment variable, and is most useful for interconnect specifications. A collection of variables is available that tells Platform MPI which interconnects to search for and which libraries and modules to look for with each interconnect. These environment variables are the primary use of `pcmpi.conf`.

Syntactically, single and double quotes in `pcmpi.conf` can be used to create values containing spaces. If a value containing a quote is needed, two adjacent quotes are interpreted as a quote to be included in the value. When not contained in quotes, spaces are interpreted as element separators in a list, and are stored as tabs.

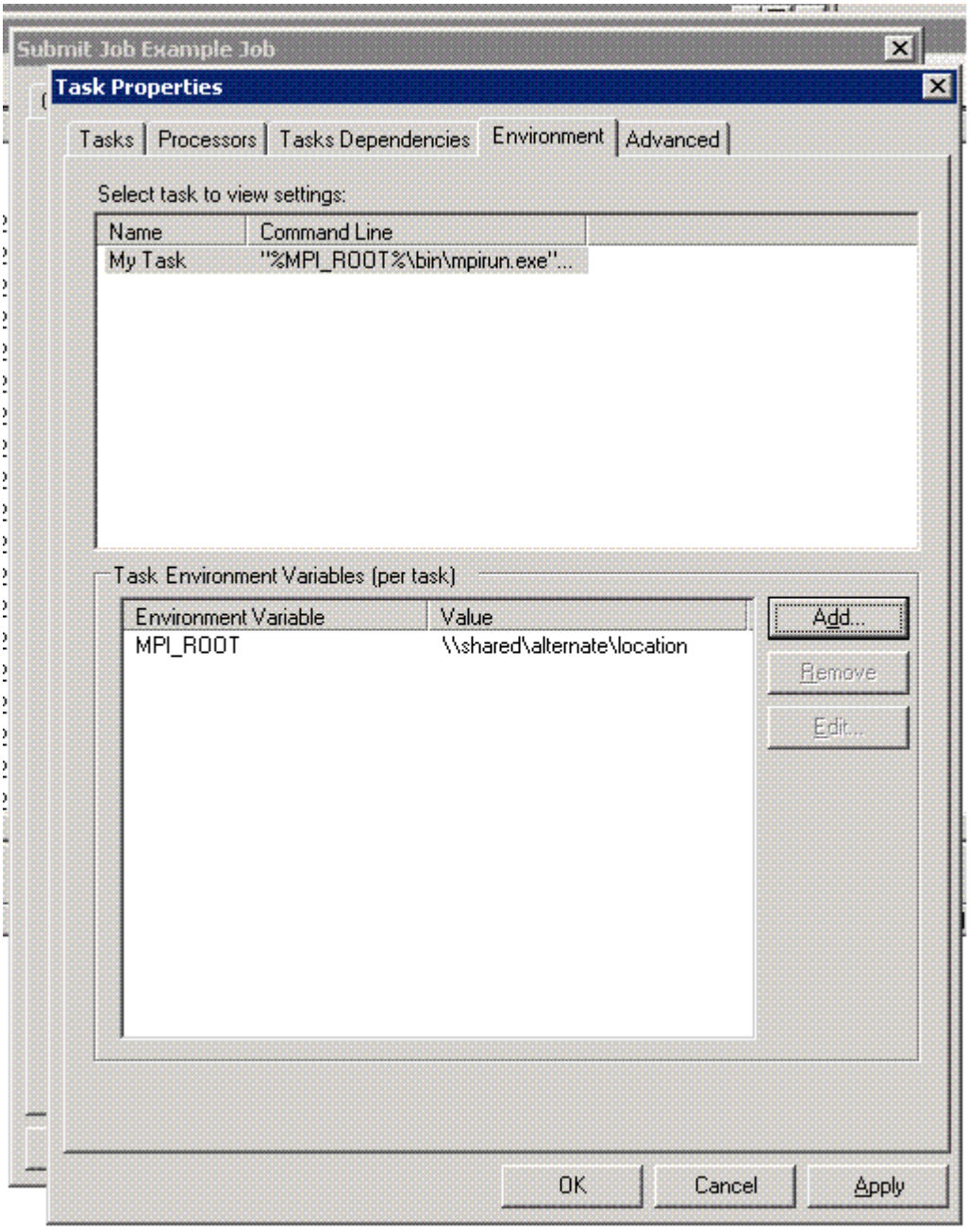
Note:

This explanation of the `pcmpi.conf` file is provided only for awareness that this functionality is available. Making changes to the `pcmpi.conf` file without contacting Platform MPI support is strongly discouraged.

Setting environment variables on Windows for HPC jobs

For Windows HPC jobs, environment variables can be set from the GUI or on the command line.

From the GUI, use the Task Properties window, Environment tab to set an environment variable.



Note:

These environment variables should be set on the **mpirun** task.

Environment variables can also be set using the flag /env. For example:

> **job add *JOBID* /numprocessors:1 /env:"MPI_ROOT=\\shared\alternate\location" ...**

List of runtime environment variables

The environment variables that affect the behavior of Platform MPI at run time are described in the following sections categorized by the following functions:

- General
- CPU bind
- Miscellaneous
- Interconnect
- InfiniBand
- Memory usage
- Connection related
- RDMA
- prun/srun
- TCP
- Elan
- Rank ID

General environment variables

MPIRUN_OPTIONS

MPIRUN_OPTIONS is a mechanism for specifying additional command-line arguments to `mpirun`. If this environment variable is set, the `mpirun` command behaves as if the arguments in *MPIRUN_OPTIONS* had been specified on the `mpirun` command line. For example:

```
% export MPIRUN_OPTIONS="-v -prot"
% $MPI_ROOT/bin/mpirun -np 2 /path/to/program.x
```

is equivalent to running:

```
% $MPI_ROOT/bin/mpirun -v -prot -np 2 /path/to/program.x
```

When settings are supplied on the command line, in the *MPIRUN_OPTIONS* variable, and in an `pcmpi.conf` file, the resulting command functions as if the `pcmpi.conf` settings had appeared first, followed by the *MPIRUN_OPTIONS*, followed by the command line. Because the settings are parsed left to right, this means the later settings have higher precedence than the earlier ones.

MPI_FLAGS

`MPI_FLAGS` modifies the general behavior of Platform MPI. The `MPI_FLAGS` syntax is a comma-separated list as follows:

```
[edde,][exdb,][egdb,][eadb,][ewdb,][l,][f,][i,] [s[a|p][#],][y[#],][o,]
[+E2,][C,][D,][E,][T,][z]
```

The following is a description of each flag:

edde

Starts the application under the dde debugger. The debugger must be in the command search path.

exdb

Starts the application under the xdb debugger. The debugger must be in the command search path.

egdb

Starts the application under the gdb debugger. The debugger must be in the command search path.

eadb

Starts the application under adb: the absolute debugger. The debugger must be in the command search path.

ewdb

Starts the application under the wdb debugger. The debugger must be in the command search path.

epathdb

Starts the application under the path debugger. The debugger must be in the command search path.

l

Reports memory leaks caused by not freeing memory allocated when a Platform MPI job is run. For example, when you create a communicator or user-defined datatype after you call `MPI_Init`, you must free the memory allocated to these objects before you call `MPI_Finalize`. In C, this is analogous to making calls to malloc() and free() for each object created during program execution.

Setting the `l` option can decrease application performance.

f

Forces MPI errors to be fatal. Using the `f` option sets the *MPI_ERRORS_ARE_FATAL* error handler, ignoring the programmer's choice of error handlers. This option can help you detect nondeterministic error problems in your code.

If your code has a customized error handler that does not report that an MPI call failed, you will not know that a failure occurred. Thus your application could be catching an error with a user-written error handler (or with *MPI_ERRORS_RETURN*) that masks a problem.

i

Turns on language interoperability for the *MPI_BOTTOM* constant.

MPI_BOTTOM Language Interoperability : Previous versions of Platform MPI were not compliant with Section 4.12.6.1 of the MPI-2 Standard which requires that sends/receives based at *MPI_BOTTOM* on a data type created with absolute addresses must access the same data regardless of the language in which the data type was created. For compliance with the standard, set *MPI_FLAGS*=i to turn on language interoperability for the *MPI_BOTTOM* constant. Compliance with the standard can break source compatibility with some MPICH code.

s[a|p][#]

Selects signal and maximum time delay for guaranteed message progression. The `sa` option selects `SIGALRM`. The `sp` option selects `SIGPROF`. The # option is the number of seconds to wait before issuing a signal to trigger message progression. The default value for the MPI library is `sp0`, which never issues a progression related signal. If the application uses both signals for its own purposes, you cannot enable the heartbeat signals.

This mechanism can be used to guarantee message progression in applications that use nonblocking messaging requests followed by prolonged periods of time in which Platform MPI routines are not called.

Generating a UNIX signal introduces a performance penalty every time the application processes are interrupted. As a result, some applications might benefit from it, others might experience a decrease in performance. As part of tuning the performance of an application, you can control the behavior of the heartbeat signals by changing their time period or by turning them off. This is accomplished by setting the time period of the `s` option in the *MPI_FLAGS* environment variable (for example: s600). Time is in seconds.

You can use the `s[a][p]#` option with the thread-compliant library as well as the standard non thread-compliant library. Setting `s[a][p]#` for the thread-compliant library has the same effect as setting *MPI_MT_FLAGS=ct* when you use a value greater than 0 for #. The default value for the thread-compliant library is `sp0`. *MPI_MT_FLAGS=ct* takes priority over the default *MPI_FLAGS=sp0*.

Set *MPI_FLAGS=sa1* to guarantee that MPI_Cancel works for canceling sends.

To use gprof on HP XC systems, set these environment variables:

MPI_FLAGS=s0

GMON_OUT_PREFIX=/tmp/app/name

These options are ignored on Platform MPI for Windows.

y[#]

Enables spin-yield logic. # is the spin value and is an integer between zero and 10,000. The spin value specifies the number of milliseconds a process should block waiting for a message before yielding the CPU to another process.

How you apply spin-yield logic depends on how well synchronized your processes are. For example, if you have a process that wastes CPU time blocked, waiting for messages, you can use spin-yield to ensure that the process relinquishes the CPU to other processes. Do this in your appfile, by setting `y[#]` to `y0` for the process in question. This specifies zero milliseconds of spin (that is, immediate yield).

If you are running an application stand-alone on a dedicated system, the default setting *MPI_FLAGS=y* allows MPI to busy spin, improving latency. To avoid unnecessary CPU consumption when using more ranks than cores, consider using a setting such as *MPI_FLAGS=y40*.

Specifying y without a spin value is equivalent to *MPI_FLAGS=y10000*, which is the default.

Note:

Except when using `srun` or `prun` to launch, if the ranks under a single mpid exceed the number of CPUs on the node and a value of *MPI_FLAGS=y* is not specified, the default is changed to *MPI_FLAGS=y0*.

If the time a process is blocked waiting for messages is short, you can possibly improve performance by setting a spin value (between 0 and 10,000) that ensures the process does not relinquish the CPU until after the message is received, thereby reducing latency.

The system treats a nonzero spin value as a recommendation only. It does not guarantee that the value you specify is used.

o

Writes an optimization report to stdout. MPI_Cart_create and MPI_Graph_create optimize the mapping of processes onto the virtual topology only if rank reordering is enabled (set reorder=1).

In the declaration statement below, see reorder=1

```
int numtasks, rank, source, dest, outbuf, i, tag=1, inbuf[4]=
{MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,}, nbrs[4],
dims[2]={4,4}, periods[2]={0,0}, reorder=1, coords[2];
```

For example:

```
/opt/platform_mpi/bin/mpirun -np 16  -e MPI_FLAGS=o ./a.out
Reordering ranks for the call
MPI_Cart_create(comm(size=16), ndims=2,
                dims=[4 4], periods=[false false],
reorder=true)
Default mapping of processes would result communication paths
         between hosts                       :    0
         between subcomplexes                :    0
         between hypernodes                  :    0
         between CPUs within a hypernode/SMP:   24
Reordered mapping results communication paths
         between hosts                       :    0
         between subcomplexes                :    0
         between hypernodes                  :    0
         between CPUs within a hypernode/SMP:   24
Reordering will not reduce overall communication cost.
Void the optimization and adopted unreordered mapping.
rank= 2 coords= 0 2  neighbors(u,d,l,r)= -1 6 1 3
rank= 0 coords= 0 0  neighbors(u,d,l,r)= -1 4 -1 1
rank= 1 coords= 0 1  neighbors(u,d,l,r)= -1 5 0 2
rank= 10 coords= 2 2  neighbors(u,d,l,r)= 6 14 9 11
rank= 2                 inbuf(u,d,l,r)= -1 6 1 3
rank= 6 coords= 1 2  neighbors(u,d,l,r)= 2 10 5 7
rank= 7 coords= 1 3  neighbors(u,d,l,r)= 3 11 6 -1
rank= 4 coords= 1 0  neighbors(u,d,l,r)= 0 8 -1 5
rank= 0                 inbuf(u,d,l,r)= -1 4 -1 1
rank= 5 coords= 1 1  neighbors(u,d,l,r)= 1 9 4 6
rank= 11 coords= 2 3  neighbors(u,d,l,r)= 7 15 10 -1
rank= 1                 inbuf(u,d,l,r)= -1 5 0 2
rank= 14 coords= 3 2  neighbors(u,d,l,r)= 10 -1 13 15
rank= 9 coords= 2 1  neighbors(u,d,l,r)= 5 13 8 10
rank= 13 coords= 3 1  neighbors(u,d,l,r)= 9 -1 12 14
rank= 15 coords= 3 3  neighbors(u,d,l,r)= 11 -1 14 -1
rank= 10                 inbuf(u,d,l,r)= 6 14 9 11
rank= 12 coords= 3 0  neighbors(u,d,l,r)= 8 -1 -1 13
rank= 8 coords= 2 0  neighbors(u,d,l,r)= 4 12 -1 9
rank= 3 coords= 0 3  neighbors(u,d,l,r)= -1 7 2 -1
```

```
rank= 6                 inbuf(u,d,l,r)= 2 10 5 7
rank= 7                 inbuf(u,d,l,r)= 3 11 6 -1
rank= 4                 inbuf(u,d,l,r)= 0 8 -1 5
rank= 5                 inbuf(u,d,l,r)= 1 9 4 6
rank= 11                 inbuf(u,d,l,r)= 7 15 10 -1
rank= 14                 inbuf(u,d,l,r)= 10 -1 13 15
rank= 9                 inbuf(u,d,l,r)= 5 13 8 10
rank= 13                 inbuf(u,d,l,r)= 9 -1 12 14
rank= 15                 inbuf(u,d,l,r)= 11 -1 14 -1
rank= 8                 inbuf(u,d,l,r)= 4 12 -1 9
rank= 12                 inbuf(u,d,l,r)= 8 -1 -1 13
rank= 3                 inbuf(u,d,l,r)= -1 7 2 -
```

+E2

Sets -1 as the value of .TRUE. and 0 as the value for .FALSE. when returning logical values from Platform MPI routines called within Fortran 77 applications.

C

Disables ccNUMA support. Allows you to treat the system as a symmetric multiprocessor. (SMP)

D

Dumps shared memory configuration information. Use this option to get shared memory values that are useful when you want to set the *MPI_SHMEMCNTL* flag.

E[on|off]

Turns function parameter error checking on or off (the default). Checking can be turned on by the setting *MPI_FLAGS=Eon*.

T

Prints the user and system times for each MPI rank.

z

Enables zero-buffering mode. Set this flag to convert MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without rewriting your code.

MPI_MT_FLAGS

MPI_MT_FLAGS controls run-time options when you use the thread-compliant version of Platform MPI. The *MPI_MT_FLAGS* syntax is a comma-separated list as follows:

```
[ct,][single,][fun,][serial,][mult]
```

The following is a description of each flag:

ct

Creates a hidden communication thread for each rank in the job. When you enable this option, do not oversubscribe your system. For example, if you enable `ct` for a 16-process application running on a 16-way machine, the result is a 32-way job.

single

Asserts that only one thread executes.

fun

Asserts that a process can be multithreaded, but only the main thread makes MPI calls (that is, all calls are funneled to the main thread).

serial

Asserts that a process can be multithreaded, and multiple threads can make MPI calls, but calls are serialized (that is, only one call is made at a time).

mult

Asserts that multiple threads can call MPI at any time with no restrictions.

Setting *MPI_MT_FLAGS=ct* has the same effect as setting *MPI_FLAGS=s[a][p]#*, when the value of # that is greater than 0. *MPI_MT_FLAGS=ct* takes priority over the default *MPI_FLAGS=sp0* setting.

The `single`, `fun`, `serial`, and `mult` options are mutually exclusive. For example, if you specify the `serial` and `mult` options in *MPI_MT_FLAGS*, only the last option specified is processed (in this case, the `mult` option). If no run-time option is specified, the default is `mult`.

MPI_ROOT

MPI_ROOT indicates the location of the Platform MPI tree. If you move the Platform MPI installation directory from its default location in `/opt/platform_mpi` for Linux, set the *MPI_ROOT* environment variable to point to the new location.

MPI_WORKDIR

*MPI_WORKDIR*changes the execution directory. This variable is ignored when `srun` or `prun` is used.

CPU bind environment variables

MPI_BIND_MAP

MPI_BIND_MAP allows specification of the integer CPU numbers, ldom numbers, or CPU masks. These are a list of integers separated by commas (,).

MPI_CPU_AFFINITY

MPI_CPU_AFFINITY is an alternative method to using `-cpu_bind` on the command line for specifying binding strategy. The possible settings are LL, RANK, MAP_CPU, MASK_CPU, LDOM, CYCLIC, BLOCK, RR, FILL, PACKED, SLURM, and MAP_LDOM.

MPI_CPU_SPIN

MPI_CPU_SPIN allows selection of spin value. The default is 2 seconds.

MPI_FLUSH_FCACHE

MPI_FLUSH_FCACHE clears the file-cache (buffer-cache). If you add -e MPI_FLUSH_FCACHE[=x] to the `mpirun` command line, the file-cache is flushed before the code starts (where =x is an optional percent of memory at which to flush). If the memory in the file-cache is greater than x, the memory is flushed. The default value is 0 (in which case a flush is always performed). Only the lowest rank # on each host flushes the file-cache; limited to one flush per host/job.

Setting this environment variable saves time if, for example, the file-cache is currently using 8% of the memory and =x is set to 10. In this case, no flush is performed.

Example output:

```
MPI_FLUSH_FCACHE set to 0, fcache pct = 22, attempting to flush fcache on host
opteron2
```

```
MPI_FLUSH_FCACHE set to 10, fcache pct = 3, no fcache flush required on host
opteron2
```

Memory is allocated with `mmap`, then it is deallocated with `munmap` afterwards.

Miscellaneous environment variables

MPI_2BCOPY

Point-to-point bcopy() is disabled by setting *MPI_2BCOPY* to 1. Valid on PA-RISC only.

MPI_MAX_WINDOW

MPI_MAX_WINDOW is used for one-sided applications. It specifies the maximum number of windows a rank can have at the same time. It tells Platform MPI to allocate enough table entries. The default is 5.

```
% export MPI_MAX_WINDOW=10
```

The above example allows 10 windows to be established for one-sided communication.

Diagnostic/debug environment variables

MPI_DLIB_FLAGS

MPI_DLIB_FLAGS controls run-time options when you use the diagnostics library. The *MPI_DLIB_FLAGS* syntax is a comma-separated list as follows:

`[ns,][h,][strict,][nmsg,][nwarn,][dump:`*`prefix`*`,][dumpf:`*`prefix`*`][x`*`NUM`*`]`

where

ns

Disables message signature analysis.

h

Disables default behavior in the diagnostic library that ignores user-specified error handlers. The default considers all errors to be fatal.

strict

Enables MPI object-space corruption detection. Setting this option for applications that make calls to routines in the MPI-2 standard can produce false error messages.

nmsg

Disables detection of multiple buffer writes during receive operations and detection of send buffer corruptions.

nwarn

Disables the warning messages that the diagnostic library generates by default when it identifies a receive that expected more bytes than were sent.

dump:*prefix*

Dumps (unformatted) sent and received messages to *prefix*.msgs.*rank* where rank is the rank of a specific process.

dumpf:*prefix*

Dumps (formatted) sent and received messages to *prefix*.msgs.*rank* where rank is the rank of a specific process.

x*NUM*

Defines a type-signature packing size. *NUM* is an unsigned integer that specifies the number of signature leaf elements. For programs with diverse derived datatypes the default value may be too small. If *NUM* is too small, the diagnostic library issues a warning during the MPI_Finalize operation.

MPI_ERROR_LEVEL

Controls diagnostic output and abnormal exit processing for application debugging where

0

Standard rank label text and abnormal exit processing. (Default)

1

Adds hostname and process id to rank label.

2

Adds hostname and process id to rank label. Also attempts to generate core file on abnormal exit.

MPI_NOBACKTRACE

On PA-RISC systems, a stack trace is printed when the following signals occur in an application:

- SIGILL
- SIGBUS
- SIGSEGV
- SIGSYS

If a signal is not caught by a user signal handler, Platform MPI shows a brief stack trace that can be used to locate the signal in the code.

```
Signal 10: bus error                                          PROCEDURE TRACEBACK:
(0)   0x0000489c   bar + 0xc   [./a.out]
(1)   0x000048c4   foo + 0x1c   [,/,/a.out]
(2)   0x000049d4   main + 0xa4   [./a.out]
(3)   0xc013750c   _start + 0xa8   [/usr/lib/libc.2]
(4)   0x0003b50    $START$ + 0x1a0   [./a.out]
```

This feature can be disabled for an individual signal handler by declaring a user-level signal handler for the signal. To disable for all signals, set the environment variable *MPI_NOBACKTRACE*:

```
% setenv MPI_NOBACKTRACE
```

MPI_INSTR

MPI_INSTR enables counter instrumentation for profiling Platform MPI applications. The *MPI_INSTR* syntax is a colon-separated list (no spaces between options) as follows:

prefix`[:l][:nc][:off][:api]`

where

prefix

Specifies the instrumentation output file prefix. The rank zero process writes the application's measurement data to *prefix*.instr in ASCII. If the prefix does not represent an absolute pathname, the instrumentation output file is opened in the working directory of the rank zero process when MPI_Init is called.

l

Locks ranks to CPUs and uses the CPU's cycle counter for less invasive timing. If used with gang scheduling, the :l is ignored.

nc

Specifies no clobber. If the instrumentation output file exists, MPI_Init aborts.

off

Specifies that counter instrumentation is initially turned off and only begins after all processes collectively call MPIHP_Trace_on.

api

The api option to *MPI_INSTR* collects and prints detailed information about the MPI Application Programming Interface (API). This option prints a new section in the instrumentation output file for each MPI routine called by each rank. It contains the MPI datatype and operation requested, along with message size, call counts, and timing information.

The following is sample output from -i <file>:api on the example `compute_pi.f`:

```
############################################################             api
##                         ##             api
##  Detailed  MPI_Reduce  routine information   ##             api
##                         ##             api
############################################################             api
api
-------------------------------------------------------------------------------- api
Rank MPI_Op MPI_Datatype Num Calls Contig Non-Contig Message Sizes Total Bytes api
-------------------------------------------------------------------------------- api
R: 0 sum  fortran double-precision 1     1     0  (8 - 8)    8  api
api
Num Calls  Message Sizes   Total Bytes   Time(ms)   Bytes / Time(s)  api
---------- ----------------- ------------ --------------- ----------------  api
1      [0..64]      8        1      0.008     api
api
api
-------------------------------------------------------------------------------- api
Rank MPI_Op MPI_Datatype Num Calls Contig Non-Contig Message Sizes Total Bytes  api
-------------------------------------------------------------------------------- api
R: 1 sum  fortran double-precision 1     1     0  (8 - 8)    8   api
api
Num Calls  Message Sizes   Total Bytes   Time(ms)   Bytes / Time(s)  api
---------- ----------------- ------------ --------------- ----------------  api
1      [0..64]      8        0      0.308     api
api
```

Lightweight instrumentation can be turned on by using either the -i option to `mpirun` or by setting the environment variable *MPI_INSTR*.

Instrumentation data includes some information on messages sent to other MPI worlds formed using MPI_Comm_accept(), MPI_Comm_connect(), or MPI_Comm_join(). All off-world message data is

accounted together using the designation `offw` regardless of which off-world rank was involved in the communication.

Platform MPI provides an API that enables users to access the lightweight instrumentation data on a per-process basis before the application calling MPI_Finalize(). The following declaration in C is necessary to access this functionality:

extern int hpmp_instrument_runtime(int reset)

A call to hpmp_instrument_runtime(0) populates the output file specified by the -i option to `mpirun` or the *MPI_INSTR* environment variable with the statistics available at the time of the call. Subsequent calls to hpmp_instrument_runtime() or MPI_Finalize() will overwrite the contents of the specified file. A call to hpmp_instrument_runtime(1) populates the file in the same way, but also resets the statistics. If instrumentation is not being used, the call to hpmp_instrument_runtime() has no effect.

Even though you can specify profiling options through the *MPI_INSTR* environment variable, the recommended approach is to use the `mpirun` command with the -i option instead. Using `mpirun` to specify profiling options guarantees that multihost applications do profiling in a consistent manner.

Counter instrumentation and trace-file generation are mutually exclusive profiling techniques.

Note:

When you enable instrumentation for multihost runs, and invoke `mpirun` on a host where an MPI process is running, or on a host remote from all MPI processes, Platform MPI writes the instrumentation output file (*prefix*.instr) to the working directory on the host that is running rank 0 or the lowest rank remaining if -ha is used.

TOTALVIEW

When you use the TotallView debugger, Platform MPI uses your *PATH* variable to find TotalView. You can also set the absolute path and TotalView options in the *TOTALVIEW* environment variable. This environment variable is used by `mpirun`.

```
% setenv TOTALVIEW /opt/totalview/bin/totalview
```

Interconnect selection environment variables

MPI_IC_ORDER

MPI_IC_ORDER is an environment variable whose default contents are "ibv:vapi:udapl:psm:mx:gm:elan:itapi:TCP" and instructs Platform MPI to search in a specific order for the presence of an interconnect. Lowercase selections imply use if detected; otherwise, keep searching. An uppercase option demands that the interconnect option be used. if it cannot be selected the application terminates with an error. For example:

```
% export MPI_IC_ORDER="ibv:vapi:udapl:psm:mx:gm:elan: itapi:TCP"
% export MPIRUN_OPTIONS="-prot"
% $MPI_ROOT/bin/mpirun -srun -n4 ./a.out
```

The command line for the above appears to `mpirun` as `$MPI_ROOT/bin/mpirun -prot -srun -n4 ./a.out` and the interconnect decision looks for the presence of Elan and uses it if found. Otherwise, interconnects are tried in the order specified by *MPI_IC_ORDER*.

The following is an example of using TCP over GigE, assuming GigE is installed and 192.168.1.1 corresponds to the Ethernet interface with GigE. The implicit use of -netaddr 192.168.1.1 is required to effectively get TCP over the proper subnet.

```
% export MPI_IC_ORDER="ibv:vapi:udapl:psm:mx:gm:elan: itapi:TCP"
% export MPIRUN_SYSTEM_OPTIONS="-netaddr 192.168.1.1"
% $MPI_ROOT/bin/mpirun -prot -TCP -srun -n4 ./a.out
```

MPI_IC_SUFFIXES

When Platform MPI is determining the availability of a given interconnect on Linux, it tries to open libraries and find loaded modules based on a collection of variables.

The use of interconnect environment variables *MPI_ICLIB_ELAN*, *MPI_ICLIB_GM*, *MPI_ICLIB_ITAPI*, *MPI_ICLIB_MX*, *MPI_ICLIB_UDAPL*, *MPI_ICLIB_VAPI*, and *MPI_ICLIB_VAPIDIR* has been deprecated.

MPI_COMMD

MPI_COMMD routes all off-host communication through daemons rather than between processes. The `MPI_COMMD` syntax is as follows:

`out_frags,in_frags`

where

out_frags

Specifies the number of 16 KB fragments available in shared memory for outbound messages. Outbound messages are sent from processes on a given host to processes on other hosts using the communication daemon.

The default value for *out_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

in_frags

Specifies the number of 16 KB fragments available in shared memory for inbound messages. Inbound messages are sent from processes on hosts to processes on a given host using the communication daemon.

The default value for *in_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

When -commd is used, *MPI_COMMD* specifies daemon communication fragments.

InfiniBand environment variables

MPI_IB_MULTIRAIL

Supports multi-rail on OpenFabric. This environment variable is ignored by all other interconnects. In multi-rail mode, a rank can use all the node cards, but only if its peer rank uses the same number of cards. Messages are striped among all the cards to improve bandwidth.

By default, multi-card message striping is off. Specify -e MPI_IB_MULTIRAIL=N, where N is the number of cards used by a rank. If N <= 1, then message striping is not used. If N is greater than the maximum

number of cards M on that node, then all M cards are used. If 1 < N <= M, message striping is used on N cards or less.

On a host, all ranks select all the cards in a series. For example, if there are 4 cards, and 4 ranks on that host; rank 0 uses cards 0, 1, 2, 3; rank 1 uses 1, 2, 3, 0; rank 2 uses 2, 3, 0, 1; rank 3 uses 3, 0, 1, 2. The order is important in SRQ mode because only the first card is used for short messages. But in short RDMA mode, all the cards are used in a balanced way.

MPI_IB_PORT_GID

If a cluster has multiple InfiniBand cards in each node, connected physically to separated fabrics, Platform MPI requires that each fabric has its own subnet ID. When the subnet IDs are the same, Platform MPI cannot identify which ports are on the same fabric, and the connection setup is likely to be less than desirable.

If all the fabrics have a unique subnet ID, by default Platform MPI assumes that the ports are connected based on the `ibv_devinfo` output port order on each node. All the port 1s are assumed to be connected to fabric 1, and all the port 2s are assumed to be connected to fabric 2. If all the nodes in the cluster have the first InfiniBand port connected to the same fabric with the same subnet ID, Platform MPI can run without any additional fabric topology hints.

If the physical fabric connections do not follow the convention described above, then the fabric topology information must be supplied to Platform MPI. The `ibv_devinfo -v` utility can be used on each node within the cluster to get the port GID. If all the nodes in the cluster are connected in the same way and each fabric has a unique subnet ID, the `ibv_devinfo` command only needs to be done on one node.

The *MPI_IB_PORT_GID* environment variable is used to specify which InfiniBand fabric subnet should be used by Platform MPI to make the initial InfiniBand connection between the nodes.

For example, if the user runs Platform MPI on two nodes with the following `ibv_devinfo -v` output, on the first node:

$ **ibv_devinfo -v**

```
hca_id: mthca0
fw_ver:                         4.7.0
node_guid:                      0008:f104:0396:62b4
....
max_pkeys:                      64
local_ca_ack_delay:             15
port:   1
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0000:0008:f104:0396:62b5
port:   2
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0001:0008:f104:0396:62b6
```

The following is the second node configuration:

$ **ibv_devinfo -v**

```
hca_id: mthca0
fw_ver:                           4.7.0
node_guid:                        0008:f104:0396:a56c
....
max_pkeys:                        64
local_ca_ack_delay:               15
port:   1
state:                    PORT_ACTIVE (4)
```

```
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0000:0008:f104:0396:a56d
port:   2
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0001:0008:f104:0396:a56e
```

The subnet ID is contained in the first 16 digits of the GID. The second 16 digits of the GID are the interface ID. In this example, port 1 on both nodes is on the same subnet and has the subnet prefix fe80:0000:0000:0000. By default, Platform MPI makes connections between nodes using the port 1. This port selection is only for the initial InfiniBand connection setup.

In this second example, the default connection cannot be made. The following is the first node configuration:

$ **ibv_devinfo -v**

```
hca_id: mthca0
fw_ver:                         4.7.0
node_guid:                      0008:f104:0396:62b4
....
max_pkeys:                      64
local_ca_ack_delay:             15
port:   1
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0001:0008:f104:0396:62b5
port:   2
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0000:0008:f104:0396:62b6
```

The following is the second node configuration:

$ **ibv_devinfo -v**

```
hca_id: mthca0
  fw_ver:                         4.7.0
  node_guid:                      0008:f104:0396:6270
....
max_pkeys:                      64
local_ca_ack_delay:             15
port:   1
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0000:0008:f104:0396:6271
port:   2
state:                  PORT_ACTIVE (4)
max_mtu:                2048 (4)
....
phys_state:             LINK_UP (5)
GID[  0]:               fe80:0000:0000:0001:0008:f104:0396:6272
```

In this case, the subnet with prefix fe80:0000:0000:0001 includes port 1 on the first node and port 2 on the second node. The second subnet with prefix fe80:0000:0000:0000 includes port 2 on the first node and port 1 on the second.

To make the connection using the fe80:0000:0000:0001 subnet, pass this option ot `mpirun`:

-e MPI_IB_PORT_GID=fe80:0000:0000:0001

If the *MPI_IB_PORT_GID* environment variable is not supplied to `mpirun`, Platform MPI checks the subnet prefix for the first port it chooses, determines that the subnet prefixes do not match, prints the following message, and exits:

```
pp.x: Rank 0:1: MPI_Init: The IB ports chosen for IB connection  setup do not have the
same subnet_prefix.  Please provide a port GID that all nodes have IB path  to it by
MPI_IB_PORT_GID  pp.x: Rank 0:1: MPI_Init: You can get port GID using  'ibv_devinfo -v'
```

MPI_IB_CARD_ORDER

Defines mapping of ranks to IB cards.

% **setenv MPI_IB_CARD_ORDER <*card*#>[:*port*#]**

where

card#

Ranges from 0 to N-1

port#

Ranges from 0 to 1

Card:port can be a comma-separated list that drives the assignment of ranks to cards and ports in the cards.

Platform MPI numbers the ports on a card from 0 to N-1; whereas utilities such as `vstat` display ports numbered 1 to N.

Examples:

To use the second IB card:

% **mpirun -e MPI_IB_CARD_ORDER=1 ...**

To use the second port of the second card:

% **mpirun -e MPI_IB_CARD_ORDER=1:1 ...**

To use the first IB card:

% **mpirun -e MPI_IB_CARD_ORDER=0 ...**

To assign ranks to multiple cards:

% **mpirun -e MPI_IB_CARD_ORDER=0,1,2**

This assigns the local ranks per node in order to each card.

% **mpirun -hostlist "host0 4 host1 4"**

This creates ranks 0-3 on host 0 and ranks 4-7 on host 1. It assigns rank 0 to card 0, rank 1 to card 1, rank 2 to card 2, rank 3 to card 0 all on host 0. It also assigns rank 4 to card 0, rank 5 to card 1, rank 6 to card 2, rank 7 to card 0 all on host 1.

% **mpirun -hostlist -np 8 "host0 host1"**

This creates ranks 0 through 7 alternating on host 0, host 1, host 0, host 1, etc. It assigns rank 0 to card 0, rank 2 to card 1, rank 4 to card 2, rank 6 to card 0 all on host 0. It assigns rank 1 to card 0, rank 3 to card 1, rank 5 to card 2, rank 7 to card 0 all on host 1.

MPI_IB_PKEY

Platform MPI supports IB partitioning via Mellanox VAPI and OFED Verbs API.

By default, Platform MPI searches the unique full membership partition key from the port partition key table used. If no such pkey is found, an error is issued. If multiple pkeys are found, all related pkeys are printed and an error message is issued.

If the environment variable *MPI_IB_PKEY* has been set to a value, in hex or decimal, the value is treated as the pkey and the pkey table is searched for the same pkey. If the pkey is not found, an error message is issued.

When a rank selects a pkey to use, a verification is made to make sure all ranks are using the same pkey. If ranks are not using the same pkey, and error message is issued.

MPI_IBV_QPPARAMS

MPI_IBV_QPPARAMS=a,b,c,d,e

Specifies QP settings for IBV where:

a

Time-out value for IBV retry if there is no response from target. Minimum is 1. Maximum is 31. Default is 18.

b

The retry count after a time-out before an error is issued. Minimum is 0. Maximum is 7. Default is 7.

c

The minimum Receiver Not Ready (RNR) NAK timer. After this time, an RNR NAK is sent back to the sender. Values: 1(0.01ms) - 31(491.52ms); 0(655.36ms). The default is 24(40.96ms).

d

RNR retry count before an error is issued. Minimum is 0. Maximum is 7. Default is 7 (infinite).

e

The max inline data size. Default is 128 bytes.

MPI_VAPI_QPPARAMS

MPI_VAPI_QPPARAMS=a,b,c,d

Specifies time-out setting for VAPI where:

a

Time out value for VAPI retry if there is no response from the target. Minimum is 1. Maximum is 31. Default is 18.

b

The retry count after a time-out before an error is issued. Minimum is 0. Maximum is 7. Default is 7.

c

The minimum Receiver Not Ready (RNR) NAK timer. After this time, an RNR NAK is set back to the sender. Values: 1(0.01ms) - 31(491.52ms); 0(655.36ms). The default is 24(40.96ms).

d

RNR retry count before an error is issued. Minimum is 0. Maximum is 7. Default is 7 (infinite).

Memory usage environment variables

MPI_GLOBMEMSIZE

MPI_GLOBMEMSIZE=e

Where e is the total bytes of shared memory of the job. If the job size is N, each rank has e/N bytes of shared memory. 12.5% is used as generic. 87.5% is used as fragments. The only way to change this ratio is to use *MPI_SHMEMCNTL*.

MPI_NO_MALLOCLIB

When set, *MPI_NO_MALLOCLIB* avoids using Platform MPI's ptmalloc implementation and instead uses the standard libc implementation (or perhaps a malloc implementation contained in the application).

MPI_PAGE_ALIGN_MEM

MPI_PAGE_ALIGN_MEM causes the Platform MPI library to page align and page pad memory. This is for multithreaded InfiniBand support.

% **export MPI_PAGE_ALIGN_MEM=1**

MPI_PHYSICAL_MEMORY

MPI_PHYSICAL_MEMORY allows the user to specify the amount of physical memory in MB available on the system. MPI normally attempts to determine the amount of physical memory for the purpose of determining how much memory to pin for RDMA message transfers on InfiniBand and Myrinet GM. The value determined by Platform MPI can be displayed using the -dd option. If Platform MPI specifies an incorrect value for physical memory, this environment variable can be used to specify the value explicitly:

% **export MPI_PHYSICAL_MEMORY=1024**

The above example specifies that the system has 1 GB of physical memory.

MPI_PIN_PERCENTAGE and *MPI_PHYSICAL_MEMORY* are ignored unless InfiniBand or Myrinet GM is in use.

MPI_RANKMEMSIZE

MPI_RANKMEMSIZE=d

Where d is the total bytes of shared memory of the rank. Specifies the shared memory for each rank. 12.5% is used as generic. 87.5% is used as fragments. The only way to change this ratio is to use *MPI_SHMEMCNTL*. *MPI_RANKMEMSIZE* differs from *MPI_GLOBMEMSIZE*, which is the total shared memory across all ranks on the host. *MPI_RANKMEMSIZE* takes precedence over *MPI_GLOBMEMSIZE* if both are set. *MPI_RANKMEMSIZE* and *MPI_GLOBMEMSIZE* are mutually

exclusive to *MPI_SHMEMCNTL*. If *MPI_SHMEMCNTL* is set, the user cannot set the other two, and vice versa.

MPI_PIN_PERCENTAGE

MPI_PIN_PERCENTAGE communicates the maximum percentage of physical memory (see *MPI_PHYSICAL_MEMORY*) that can be pinned at any time. The default is 20%.

```
% export MPI_PIN_PERCENTAGE=30
```

The above example permits the Platform MPI library to pin (lock in memory) up to 30% of physical memory. The pinned memory is shared between ranks of the host that were started as part of the same `mpirun` invocation. Running multiple MPI applications on the same host can cumulatively cause more than one application's *MPI_PIN_PERCENTAGE* to be pinned. Increasing *MPI_PIN_PERCENTAGE* can improve communication performance for communication-intensive applications in which nodes send and receive multiple large messages at a time, which is common with collective operations. Increasing *MPI_PIN_PERCENTAGE* allows more large messages to be progressed in parallel using RDMA transfers; however, pinning too much physical memory can negatively impact computation performance. *MPI_PIN_PERCENTAGE* and *MPI_PHYSICAL_MEMORY* are ignored unless InfiniBand or Myrinet GM is in use.

MPI_SHMEMCNTL

MPI_SHMEMCNTL controls the subdivision of each process's shared memory for point-to-point and collective communications. It cannot be used with *MPI_GLOBMEMSIZE*. The *MPI_SHMEMCNTL* syntax is a comma-separated list as follows:

nenv, *frag*, *generic*

where

nenv

Specifies the number of envelopes per process pair. The default is 8.

frag

Denotes the size in bytes of the message-passing fragments region. The default is 87.5% of shared memory after mailbox and envelope allocation.

generic

Specifies the size in bytes of the generic-shared memory region. The default is 12.5% of shared memory after mailbox and envelope allocation. The generic region is typically used for collective communication.

MPI_SHMEMCNTL=a,b,c

where

a

The number of envelopes for shared memory communication. The default is 8.

b

The bytes of shared memory to be used as fragments for messages.

c

The bytes of shared memory for other generic use, such as MPI_Alloc_mem() call.

MPI_USE_MALLOPT_AVOID_MMAP

Instructs the underlying malloc implementation to avoid mmaps and instead use sbrk() to get all memory used. The default is *MPI_USE_MALLOPT_AVOID_MMAP=0*.

Connection related environment variables

MPI_LOCALIP

MPI_LOCALIP specifies the host IP address assigned throughout a session. Ordinarily, `mpirun` determines the IP address of the host it is running on by calling gethostbyaddr. However, when a host uses SLIP or PPP, the host's IP address is dynamically assigned only when the network connection is established. In this case, gethostbyaddr might not return the correct IP address.

The *MPI_LOCALIP* syntax is as follows:

`xxx.xxx.xxx.xxx`

where *`xxx.xxx.xxx.xxx`* specifies the host IP address.

MPI_MAX_REMSH

MPI_MAX_REMSH=N

Platform MPI includes a start-up scalability enhancement when using the -f option to `mpirun`. This enhancement allows a large number of Platform MPI daemons (mpid) to be created without requiring `mpirun` to maintain a large number of remote shell connections.

When running with a very large number of nodes, the number of remote shells normally required to start all daemons can exhaust available file descriptors. To create the necessary daemons, `mpirun` uses the remote shell specified with *MPI_REMSH* to create up to 20 daemons only, by default. This number can be changed using the environment variable *MPI_MAX_REMSH*. When the number of daemons required is greater than *MPI_MAX_REMSH*, `mpirun` creates only *MPI_MAX_REMSH* number of remote daemons directly. The directly created daemons then create the remaining daemons using an n-ary tree, where n is the value of *MPI_MAX_REMSH*. Although this process is generally transparent to the user, the new start-up requires that each node in the cluster can use the specified *MPI_REMSH* command (e.g., rsh, ssh) to each node in the cluster without a password. The value of *MPI_MAX_REMSH* is used on a per-world basis. Therefore, applications that spawn a large number of worlds might need to use a small value for *MPI_MAX_REMSH*. *MPI_MAX_REMSH* is only relevant when using the -f option to `mpirun`. The default value is 20.

MPI_NETADDR

Allows control of the selection process for TCP/IP connections. The same functionality can be accessed by using the -netaddr option to `mpirun`. For more information, refer to the `mpirun` documentation.

MPI_REMSH

By default, Platform MPI attempts to use `ssh` on Linux. We recommend that `ssh` users set `StrictHostKeyChecking=no` in their `~/.ssh/config`.

To use `rsh` on Linux instead, run the following script as root on each node in the cluster:

% **/opt/pcmpi/etc/mpi.remsh.default**

Or, to use `rsh` on Linux, use the alternative method of manually populating the files `/etc/profile.d/pcmpi.csh` and `/etc/profile.d/pcmpi.sh` with the following settings respectively:

setenv MPI_REMSH rsh

export MPI_REMSH=rsh

On Linux, *MPI_REMSH* specifies a command other than the default `remsh` to start remote processes. The `mpirun`, `mpijob`, and `mpiclean` utilities support *MPI_REMSH*. For example, you can set the environment variable to use a secure shell:

% **setenv MPI_REMSH /bin/ssh**

Platform MPI allows users to specify the remote execution tool to use when Platform MPI must start processes on remote hosts. The tool must have a call interface similar to that of the standard utilities: `rsh`, `remsh` and `ssh`. An alternate remote execution tool, such as `ssh`, can be used on Linux by setting the environment variable *MPI_REMSH* to the name or full path of the tool to use:

% **export MPI_REMSH=ssh**

% **$MPI_ROOT/bin/mpirun <*options*> -f <*appfile*>**

Platform MPI also supports setting *MPI_REMSH* using the -e option to `mpirun`:

% **$MPI_ROOT/bin/mpirun -e MPI_REMSH=ssh <*options*> -f <*appfile*>**

Platform MPI also supports setting *MPI_REMSH* to a command that includes additional arguments:

% **$MPI_ROOT/bin/mpirun -e 'MPI_REMSH="ssh -x"' <*options*> -f <*appfile*>**

When using `ssh`, be sure that it is possible to use `ssh` from the host where `mpirun` is executed without `ssh` requiring interaction from the user.

RDMA tunable environment variables

MPI_RDMA_INTRALEN

-e MPI_RDMA_INTRALEN=262144

Specifies the size (in bytes) of the transition from shared memory to interconnect when -intra=mix is used. For messages less than or equal to the specified size, shared memory is used. For messages greater than that size, the interconnect is used. TCP/IP, Elan, MX, and PSM do not have mixed mode.

MPI_RDMA_MSGSIZE

MPI_RDMA_MSGSIZE=a,b,c

Specifies message protocol length where:

a

Short message protocol threshold. If the message length is bigger than this value, middle or long message protocol is used. The default is 16384 bytes.

b

Middle message protocol. If the message length is less than or equal to b, consecutive short messages are used to send the whole message. By default, b is set to 16384 bytes, the same as a, to effectively turn off middle message protocol. On IBAL, the default is 131072 bytes.

c

Long message fragment size. If the message is greater than b, the message is fragmented into pieces up to c in length (or actual length if less than c) and the corresponding piece of the user's buffer is pinned directly. The default is 4194304 bytes, but on Myrinet GM and IBAL the default is 1048576 bytes.

When deferred deregistration is used, pinning memory is fast. Therefore, the default setting for *MPI_RDMA_MSGSIZE* is 16384, 16384, 4194304 which means any message over 16384 bytes is pinned for direct use in RDMA operations.

However, if deferred deregistration is not used (-ndd), then pinning memory is expensive. In that case, the default setting for *MPI_RDMA_MSGSIZE* is 16384, 262144, 4194304 which means messages larger than 16384 and smaller than or equal to 262144 bytes are copied into pre-pinned memory using Platform MPI middle message protocol rather than being pinned and used in RDMA operations directly.

The middle message protocol performs better than the long message protocol if deferred deregistration is not used.

For more information, see the *MPI_RDMA_MSGSIZE* section of the `mpienv` manpage.

MPI_RDMA_NENVELOPE

MPI_RDMA_NENVELOPE=N

Specifies the number of short message envelope pairs for each connection if RDMA protocol is used, where N is the number of envelope pairs. The default is from 8 to 128 depending on the number of ranks.

MPI_RDMA_NFRAGMENT

MPI_RDMA_NFRAGMENT=N

Specifies the number of long message fragments that can be concurrently pinned down for each process, sending or receiving. The maximum number of fragments that can be pinned down for a process is 2*N. The default value of N is 128.

MPI_RDMA_NONESIDED

MPI_RDMA_NONESIDED=N

Specifies the number of one-sided operations that can be posted concurrently for each rank, regardless of the destination. The default is 8.

MPI_RDMA_NSRQRECV

MPI_RDMA_NSRQRECV=K

Specifies the number of receiving buffers used when the shared receiving queue is used, where K is the number of receiving buffers. If N is the number of off host connections from a rank, the default value is calculated as the smaller of the values Nx8 and 2048.

In the above example, the number of receiving buffers is calculated as 8 times the number of off host connections. If this number is greater than 2048, the maximum number used is 2048.

prun/srun environment variables

MPI_PROT_BRIEF

Disables the printing of the host name or IP address, and the rank mappings when `-prot` is specified in the `mpirun` command line.

In normal cases, that is, when all of the on-node and off-node ranks communicate using the same protocol, only two lines are displayed, otherwise, the entire matrix displays. This allows you to see when abnormal or unexpected protocols are being used.

MPI_PROT_MAX

Specifies the maximum number of columns and rows displayed in the `-prot` output table. This number corresponds to the number of mpids that the job uses, which is typically the number of hosts when block scheduling is used, but can be up to the number of ranks if cyclic scheduling is used.

Regardless of size, the `-prot` output table is always displayed when not all of the inter-node or intra-node communications use the same communication protocol.

MPI_PRUNOPTIONS

Allows `prun` specific options to be added automatically to the `mpirun` command line. For example:

% **export MPI_PRUNOPTIONS="-m cyclic -x host0"**

% **mpirun -prot -prun -n2 ./a.out**

is equivalent to:

% **mpirun -prot -prun -m cyclic -x host0 -n2 ./a.out**

MPI_SPAWN_PRUNOPTIONS

Allows `prun`options to be implicitly added to the launch command when SPAWN functionality is used to create new ranks with `prun`.

MPI_SPAWN_SRUNOPTIONS

Allows `srun` options to be implicitly added to the launch command when SPAWN functionality is used to create new ranks with `srun`.

MPI_SRUNOPTIONS

Allows additional `srun` options to be specified such as --label.

% **setenv MPI_SRUNOPTIONS <*option*>**

MPI_USEPRUN

Platform MPI provides the capability to automatically assume that `prun` is the default launching mechanism. This mode of operation automatically classifies arguments into `prun` and `mpirun` arguments and correctly places them on the command line. The assumed `prun` mode also allows appfiles to be interpreted for command-line arguments and translated into `prun`mode. The implied `prun` method of launching is useful for applications that embed or generate `mpirun` invocations deeply in the application.

MPI_USEPRUN_IGNORE_ARGS

Provides an easy way to modify the arguments contained in an appfile by supplying a list of space-separated arguments that `mpirun` should ignore.

% **setenv MPI_USEPRUN_IGNORE_ARGS <*option*>**

MPI_USESRUN

Platform MPI provides the capability to automatically assume that `srun` is the default launching mechanism. This mode of operation automatically classifies arguments into `srun` and `mpirun` arguments and correctly places them on the command line. The assumed `srun` mode also allows appfiles to be interpreted for command-line arguments and translated into `srun`mode. The implied `srun` method of launching is useful for applications that embed or generate their `mpirun` invocations deeply within the application. This allows existing ports of an application from a Platform MPI supported platform to HP XC.

MPI_USESRUN_IGNORE_ARGS

Provides an easy way to modify the arguments contained in an appfile by supplying a list of space-separated arguments that `mpirun` should ignore.

% **setenv MPI_USESRUN_IGNORE_ARGS <*option*>**

In the example below, the command line contains a reference to -stdio=bnone which is filtered out because it is set in the ignore list.

% **setenv MPI_USESRUN_VERBOSE 1**

% **setenv MPI_USESRUN_IGNORE_ARGS -stdio=bnone**

% **setenv MPI_USESRUN 1**

% **setenv MPI_SRUNOPTION --label**

% **bsub -I -n4 -ext "SLURM[nodes=4]" $MPI_ROOT/bin/mpirun -stdio=bnone -f appfile -- pingpong**

```
Job <369848> is submitted to default queue <normal>.
<<Waiting for dispatch ...>>
<<Starting on lsfhost.localdomain>>
/opt/platform_mpi/bin/mpirun unset
MPI_USESRUN;/opt/platform_mpi/bin/mpirun -srun./pallas.x -npmin 4 pingpong
```

TCP environment variables

MPI_TCP_CORECVLIMIT

The integer value indicates the number of simultaneous messages larger than 16 KB that can be transmitted to a single rank at once via TCP/IP. Setting this variable to a larger value can allow Platform MPI to use more parallelism during its low-level message transfers, but can greatly reduce performance by causing switch congestion. Setting *MPI_TCP_CORECVLIMIT* to zero does not limit the number of simultaneous messages a rank can receive at once. The default value is 0.

MPI_SOCKBUFSIZE

Specifies, in bytes, the amount of system buffer space to allocate for sockets when using TCP/IP for communication. Setting *MPI_SOCKBUFSIZE* results in calls to `setsockopt` (..., *SOL_SOCKET*,

SO_SNDBUF, ...) and `setsockopt` (..., *SOL_SOCKET*, *SO_RCVBUF*, ...). If unspecified, the system default (which on many systems is 87380 bytes) is used.

Elan environment variables

MPI_USE_LIBELAN

By default when Elan is in use, the Platform MPI library uses Elan's native collective operations for performing MPI_Bcast and MPI_ Barrier operations on *MPI_COMM_WORLD* sized communicators. To change this behavior, set *MPI_USE_LIBELAN* to false or 0. If changed, these operations are implemented using point-to-point Elan messages.

To turn off:

```
% export MPI_USE_LIBELAN=0
```

MPI_USE_LIBELAN_SUB

The use of Elan's native collective operations can be extended to include communicators that are smaller than *MPI_COMM_WORLD* by setting the *MPI_USE_LIBELAN_SUB* environment variable to a positive integer. By default, this functionality is disabled because libelan memory resources are consumed and can eventually cause run-time failures when too many subcommunicators are created.

```
% export MPI_USE_LIBELAN_SUB=10
```

MPI_ELANLOCK

By default, Platform MPI only provides exclusive window locks via Elan lock when using the Elan interconnect. To use Platform MPI shared window locks, the user must turn off Elan lock and use window locks via shared memory. In this way, exclusive and shared locks are from shared memory. To turn off Elan locks, set *MPI_ELANLOCK* to zero.

```
% export MPI_ELANLOCK=0
```

Windows HPC environment variables

MPI_SAVE_TASK_OUTPUT

Saves the output of the scheduled `HPCCPService` task to a file unique for each node. This option is useful for debugging startup issues. This option is not set by default.

MPI_FAIL_ON_TASK_FAILURE

Sets the scheduled job to fail if any task fails. The job will stop execution and report as failed if a task fails. The default is set to true (1). To turn off, set to 0.

MPI_COPY_LIBHPC

Controls when `mpirun` copies `libhpc.dll` to the first node of HPC job allocation. Values:

- 0 – Don't copy.
- 1 (default) – Use cached `libhpc` on compute node.
- 2 – Copy and overwrite cached version on compute nodes.

Rank identification environment variables

Platform MPI sets several environment variables to let the user access information about the MPI rank layout prior to calling *MPI_Init*. These variables differ from the others in this section in that the user doesn't set these to provide instructions to Platform MPI. Platform MPI sets them to give information to the user's application.

HPMPI=1

This is set so that an application can conveniently tell if it is running under Platform MPI.

MPI_NRANKS

This is set to the number of ranks in the MPI job.

MPI_RANKID

This is set to the rank number of the current process.

MPI_LOCALNRANKS

This is set to the number of ranks on the local host.

MPI_LOCALRANKID

This is set to the rank number of the current process relative to the local host (0.. *MPI_LOCALNRANKS*-1).

These settings are not available when running under `srun` or `prun`. However, similar information can be gathered from variables set by those systems, such as *SLURM_NPROCS* and *SLURM_PROCID*.

Scalability

Interconnect support of MPI-2 functionality

Platform MPI has been tested on InfiniBand clusters with as many as 2048 ranks using the VAPI protocol. Most Platform MPI features function in a scalable manner. However, the following are still subject to significant resource growth as the job size grows.

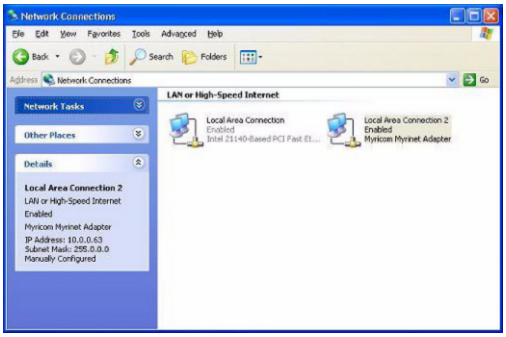
Table 17: Scalability

Feature	Affected Interconnect/ Protocol	Scalability Impact
spawn	All	Forces use of pairwise socket connections between all mpid's (typically one mpid per machine).
one-sided shared lock/ unlock	All except VAPI and IBV	Only VAPI and IBV provide low-level calls to efficiently implement shared lock/unlock. All other interconnects require mpid's to satisfy this feature.
one-sided exclusive lock/unlock	All except VAPI, IBV, and Elan	VAPI, IBV, and Elan provide low-level calls that allow Platform MPI to efficiently implement exclusive lock/ unlock. All other interconnects require mpid's to satisfy this feature.
one-sided other	TCP/IP	All interconnects other than TCP/IP allow Platform MPI to efficiently implement the remainder of the one-sided functionality. Only when using TCP/IP are mpid's required to satisfy this feature.

Resource usage of TCP/IP communication

Platform MPI has been tested on large Linux TCP/IP clusters with as many as 2048 ranks. Because each Platform MPI rank creates a socket connection to each other remote rank, the number of socket descriptors required increases with the number of ranks. On many Linux systems, this requires increasing the operating system limit on per-process and system-wide file descriptors.

The number of sockets used by Platform MPI can be reduced on some systems at the cost of performance by using daemon communication. In this case, the processes on a host use shared memory to send messages to and receive messages from the daemon. The daemon, in turn, uses a socket connection to communicate with daemons on other hosts. Using this option, the maximum number of sockets opened by any Platform MPI process grows with the number of hosts used by the MPI job rather than the number of total ranks.



To use daemon communication, specify the -commd option in the `mpirun` command. After you set the -commd option, you can use the *MPI_COMMD* environment variable to specify the number of shared-memory fragments used for inbound and outbound messages. Daemon communication can result in lower application performance. Therefore, it should only be used to scale an application to a large number of ranks when it is not possible to increase the operating system file descriptor limits to the required values.

Resource usage of RDMA communication modes

When using InfiniBand or GM, some memory is pinned, which means it is locked to physical memory and cannot be paged out. The amount of prepinned memory Platform MPI uses can be adjusted using several tunables, such as *MPI_RDMA_MSGSIZE*, *MPI_RDMA_NENVELOPE*, *MPI_RDMA_NSRQRECV*, and *MPI_RDMA_NFRAGMENT*.

By default when the number of ranks is less than or equal to 512, each rank prepins 256 Kb per remote rank; thus making each rank pin up to 128 Mb. If the number of ranks is above 512 but less than or equal to 1024, then each rank only prepins 96 Kb per remote rank; thus making each rank pin up to 96 Mb. If the number of ranks is over 1024, then the 'shared receiving queue' option is used which reduces the amount of prepinned memory used for each rank to a fixed 64 Mb regardless of how many ranks are used.

Platform MPI also has safeguard variables *MPI_PHYSICAL_MEMORY* and *MPI_PIN_PERCENTAGE* which set an upper bound on the total amount of memory a Platform MPI job will pin. An error is reported during start-up if this total is not large enough to accommodate the prepinned memory.

Dynamic processes

Platform MPI provides support for dynamic process management, specifically the spawning, joining, and connecting of new processes. MPI_Comm_spawn() starts MPI processes and establishes communication with them, returning an intercommunicator.

MPI_Comm_spawn_multiple() starts several binaries (or the same binary with different arguments), placing them in the same comm_world and returning an intercommunicator. The MPI_Comm_spawn () and MPI_Comm_spawn_multiple() routines provide an interface between MPI and the runtime environment of an MPI application.

MPI_Comm_accept() and MPI_Comm_connect() along with MPI_Open_port() and MPI_Close_port () allow two independently run MPI applications to connect to each other and combine their ranks into a single communicator.

MPI_Comm_join() allows two ranks in independently run MPI applications to connect to each other and form an intercommunicator given a socket connection between them.

Processes that are not part of the same MPI world, but are introduced through calls to MPI_Comm_connect(), MPI_Comm_accept(), MPI_Comm_spawn(), or MPI_Comm_spawn_multiple () attempt to use InfiniBand for communication. Both sides need to have InfiniBand support enabled and use the same InfiniBand parameter settings, otherwise TCP will be used for the connection. Only OFED IBV protocol is supported for these connections. When the connection is established through one of these MPI calls, a TCP connection is first established between the root process of both sides. TCP connections are set up among all the processes. Finally, IBV InfiniBand connections are established among all process pairs, and the TCP connections are closed.

Spawn functions supported in Platform MPI:

- MPI_Comm_get_parent()
- MPI_Comm_spawn()
- MPI_Comm_spawn_multiple()
- MPI_Comm_accept()
- MPI_Comm_connect()
- MPI_Open_port()
- MPI_Close_port()
- MPI_Comm_join()

Keys interpreted in the info argument to the spawn calls:

- host : We accept standard host.domain strings and start the ranks on the specified host. Without this key, the default is to start on the same host as the root of the spawn call.
- wdir : We accept `/some/directory` strings.
- path : We accept `/some/directory:/some/other/directory`.

A mechanism for setting arbitrary environment variables for the spawned ranks is not provided.

Singleton launching

Platform MPI supports the creation of a single rank without the use of `mpirun`, called singleton launching. It is only valid to launch an *MPI_COMM_WORLD* of size one using this approach. The single rank created in this way is executed as if it were created with `mpirun -np 1` <*executable*>. Platform MPI environment variables can influence the behavior of the rank. Interconnect selection can be controlled using the environment variable *MPI_IC_ORDER*. Many command-line options that would normally be passed to `mpirun` cannot be used with singletons. Examples include, but are not limited to, -cpu_bind, -d, -prot, -ndd, -srq, and -T. Some options, such as -i, are accessible through environment variables (*MPI_INSTR*) and can still be used by setting the appropriate environment variable before creating the process.

Creating a singleton using fork() and exec() from another MPI process has the same limitations that OFED places on fork() and exec().

License release/regain on suspend/resume

Platform MPI supports the release and regain of license keys when a job is suspended and resumed by a job scheduler. This feature is recommended for use only with a batch job scheduler. To enable this feature, add *HPMPI_ALLOW_LICENSE_RELEASE=1* to the `mpirun` command line. When `mpirun` receives a `SIGTSTP`, the licenses that are used for that job are released back to the license server. Those released licenses can run another Platform MPI job while the first job remains suspended. When a suspended `mpirun` job receives a `SIGCONT`, the licenses are reacquired and the job continues. If the licenses cannot be reacquired from the license server, the job exits.

When a job is suspended in Linux, any memory that is pinned is not swapped to disk, and is not handled by the operating system virtual memory subsystem. Platform MPI pins memory that is associated with RDMA message transfers. By default, up to 20% of the system memory can be pinned by Platform MPI at any one time. The amount of memory that is pinned can be changed by two environment variables: *MPI_PHYSICAL_MEMORY* and *MPI_PIN_PERCENTAGE* (default 20%). The -dd option to `mpirun` displays the amount of physical memory that is detected by Platform MPI. If the detection is wrong, the correct amount of physical memory should be set with *MPI_PHYSICAL_MEMORY* in bytes. This memory is only returned to the operating system for use by other processes after the job resumes and exits.

Improved deregistration using ptmalloc (Linux only)

To achieve the best performance on RDMA-enabled interconnects like InfiniBand and Myrinet, the MPI library must be aware when memory is returned to the system in malloc() and free() calls. To enable more robust handling of that information, Platform MPI contains a copy of the ptmalloc implementation and uses it by default.

For applications with specific needs, there are a number of available modifications to this default configuration. To avoid using Platform MPI's ptmalloc implementation and instead use the standard libc implementation (or perhaps a malloc implementation contained in the application), set the environment variable *MPI_NO_MALLOCLIB* at run time.

If the above option is applied so that the ptmalloc contained in Platform MPI is not used, there is a risk of MPI not being informed when memory is returned to the system. This can be alleviated with the settings *MPI_USE_MALLOPT_SBRK_PROTECTION* and *MPI_USE_MALLOPT_AVOID_MMAP* at run time, which essentially results in the libc malloc implementation not returning memory to the system.

There are cases where these two settings cannot keep libc from returning memory to the system, specifically when multiple threads call malloc/free at the same time. In these cases, the only remaining option is to disable Platform MPI's lazy deregistration by giving the -ndd flag to `mpirun`.

In the default case where the ptmalloc contained in Platform MPI is used, the above cases are avoided and lazy deregistration works correctly as is. So the above tunables are only recommended for applications with special requirements concerning their malloc/free usage.

Signal propagation (Linux only)

Platform MPI supports the propagation of signals from `mpirun` to application ranks. The `mpirun` executable traps the following signals and propagates them to the ranks:

SIGINT

SIGTERM

SIGABRT

SIGALRM

SIGFPE

SIGHUP

SIGILL

SIGPIPE

SIGQUIT

SIGSEGV

SIGUSR1

SIGUSR2

SIGBUS

SIGPROF

SIGSYS

SIGTRAP

SIGURG

SIGVTALRM

SIGPOLL

SIGCONT

SIGTSTP

If `prun`/`srun` is used for launching the application, then `mpirun` sends the signal to the responsible launcher and relies on the signal propagation capabilities of the launcher to ensure that the signal is propagated to the ranks. When using `prun`, SIGTTIN is also intercepted by `mpirun`, but is not propagated.

When using an appfile, Platform MPI propagates these signals to remote Platform MPI daemons (mpid) and local ranks. Each daemon propagates the signal to the ranks it created. An exception is the treatment of SIGTSTP. When a daemon receives an SIGTSTP signal, it propagates SIGSTOP to the ranks it created and then raises SIGSTOP on itself. This allows all processes related to a Platform MPI execution to be suspended and resumed using SIGTSTP and SIGCONT.

The Platform MPI library also changes the default signal-handling properties of the application in a few specific cases. When using the -ha option to `mpirun`, SIGPIPE is ignored. When using *MPI_FLAGS=U*, an MPI signal handler for printing outstanding message status is established for SIGUSR1. When using *MPI_FLAGS=sa*, an MPI signal handler used for message propagation is established for SIGALRM. When using *MPI_FLAGS=sp*, an MPI signal handler used for message propagation is established for SIGPROF.

In general, Platform MPI relies on applications terminating when they are sent SIGTERM. Applications that intercept SIGTERM might not terminate properly.

MPI-2 name publishing support

Platform MPI supports the MPI-2 dynamic process functionality MPI_Publish_name, MPI_Unpublish_name, MPI_Lookup_name, with the restriction that a separate nameserver must be started up on a server.

The service can be started as:

```
% $MPI_ROOT/bin/nameserver
```

and prints out an IP and port. When running `mpirun`, the extra option -nameserver with an IP address and port must be provided:

```
% $MPI_ROOT/bin/mpirun -spawn -nameserver <IP:port> ...
```

The scope over which names are published and retrieved consists of all `mpirun` commands that are started using the same IP:port for the nameserver.

Native language support

By default, diagnostic messages and other feedback from Platform MPI are provided in English. Support for other languages is available through the use of the Native Language Support (NLS) catalog and the internationalization environment variable NLSPATH.

The default NLS search path for Platform MPI is $NLSPATH. For NLSPATH usage, see the environ(5) manpage.

When an MPI language catalog is available, it represents Platform MPI messages in two languages. The messages are paired so that the first in the pair is always the English version of a message and the second in the pair is the corresponding translation to the language of choice.

For more information about Native Language Support, see the hpnls (5), environ (5), and lang (5) manpages.

CHAPTER 5

Profiling

This chapter provides information about utilities you can use to analyze Platform MPI applications.

Using counter instrumentation

Counter instrumentation is a lightweight method for generating cumulative run-time statistics for MPI applications. When you create an instrumentation profile, Platform MPI creates an output file in ASCII format.

You can create instrumentation profiles for applications linked with the standard Platform MPI library. For applications linked with Platform MPI V2.1 or later, you can also create profiles for applications linked with the thread-compliant library (-lmtmpi). Instrumentation is not supported for applications linked with the diagnostic library (-ldmpi).

Creating an instrumentation profile

Counter instrumentation is a lightweight method for generating cumulative run-time statistics for MPI applications. When you create an instrumentation profile, Platform MPI creates an ASCII format file containing statistics about the execution.

Instrumentation is not supported for applications linked with the diagnostic library (-ldmpi).

The syntax for creating an instrumentation profile is:

mpirun -i prefix[:l][:nc][:off]

where

prefix

Specifies the instrumentation output file prefix. The rank zero process writes the application's measurement data to *prefix*.instr in ASCII. If the prefix does not represent an absolute pathname, the instrumentation output file is opened in the working directory of the rank zero process when MPI_Init is called.

l

Locks ranks to CPUs and uses the CPU's cycle counter for less invasive timing. If used with gang scheduling, the :l is ignored.

nc

Specifies no clobber. If the instrumentation output file exists, MPI_Init aborts.

off

Specifies that counter instrumentation is initially turned off and only begins after all processes collectively call MPIHP_Trace_on.

For example, to create an instrumentation profile for an executable called `compute_pi`:

```
% $MPI_ROOT/bin/mpirun -i compute_pi -np 2 compute_pi
```

This invocation creates an ASCII file named compute_pi.instr containing instrumentation profiling.

Although -i is the preferred method of controlling instrumentation, the same functionality is also accessible by setting the *MPI_INSTR* environment variable.

Specifications you make using `mpirun -i` override specifications you make using the *MPI_INSTR* environment variable.

MPIHP_Trace_on and MPIHP_Trace_off

By default, the entire application is profiled from MPI_Init to MPI_Finalize. However, Platform MPI provides the nonstandard MPIHP_Trace_on and MPIHP_Trace_off routines to collect profile information for selected code sections only.

To use this functionality:

1. Insert the MPIHP_Trace_on and MPIHP_Trace_off pair around code that you want to profile.
2. Build the application and invoke `mpirun` with the -i <prefix> off; option. -i <index> off; specifies that counter instrumentation is enabled but initially turned off. Data collection begins after all processes collectively call MPIHP_Trace_on. Platform MPI collects profiling information only for code between MPIHP_Trace_on and MPIHP_Trace_off

Viewing ASCII instrumentation data

The ASCII instrumentation profile is a text file with the .instr extension. For example, to view the instrumentation file for the `compute_pi.f` application, you can print the *prefix*.instr file. If you defined *prefix* for the file as compute_pi, you would print compute_pi.instr.

Whether `mpirun` is invoked on a host where at least one MPI process is running or on a host remote from all MPI processes, Platform MPI writes the instrumentation output file `prefix.instr` to the working directory on the host that is running rank 0 (when instrumentation for multihost runs is enabled). When using -ha, the output file is located on the host that is running the lowest existing rank number at the time the instrumentation data is gathered during MPI_Finalize().

The ASCII instrumentation profile provides the version, the date your application ran, and summarizes information according to application, rank, and routines.

The information available in the *prefix*.instr file includes:

- Overhead time : The time a process or routine spends inside MPI (for example, the time a process spends doing message packing or spinning waiting for message arrival).
- Blocking time : The time a process or routine is blocked waiting for a message to arrive before resuming execution.

Note:

Overhead and blocking times are most useful when using -e MPI_FLAGS=y0.

- Communication hot spots : The processes in your application for which the largest amount of time is spent in communication.
- Message bin : The range of message sizes in bytes. The instrumentation profile reports the number of messages according to message length.

The following displays the contents of the example report compute_pi.instr.

ASCII Instrumentation Profile

```
Version: Platform MPI 01.08.00.00 B6060BA
Date:    Mon Apr 01 15:59:10 2002
Processes: 2
User time:   6.57%
MPI time :  93.43% [Overhead:93.43% Blocking:0.00%]
-----------------------------------------------------------------
--------------------   Instrumentation Data --------------------
-----------------------------------------------------------------
Application Summary by Rank (second):
Rank       Proc CPU Time     User Portion       System Portion
-----------------------------------------------------------------
0      0.040000        0.010000( 25.00%)     0.030000( 75.00%)
```

```
1      0.030000        0.010000( 33.33%)     0.020000( 66.67%)
-------------------------------------------------------------------
Rank       Proc Wall Time      User                  MPI
-------------------------------------------------------------------
0       0.126335     0.008332(  6.60%)       0.118003( 93.40%)
1       0.126355     0.008260(  6.54%)       0.118095( 93.46%)
-------------------------------------------------------------------
                             Rank      Proc MPI Time     Overhead            Blocking
-------------------------------------------------------------------
0        0.118003       0.118003(100.00%)    0.000000(  0.00%)
1        0.118095       0.118095(100.00%)    0.000000(  0.00%)
-------------------------------------------------------------------
Routine Summary by Rank:
Rank Routine   Statistic   Calls   Overhead(ms)   Blocking(ms)
-------------------------------------------------------------------
0
MPI_Bcast                   1        5.397081        0.000000
MPI_Finalize                1        1.238942        0.000000
MPI_Init                    1      107.195973        0.000000
MPI_Reduce                  1        4.171014        0.000000
-------------------------------------------------------------------
1
MPI_Bcast                   1        5.388021        0.000000
MPI_Finalize                1        1.325965        0.000000
MPI_Init                    1      107.228994        0.000000
MPI_Reduce                  1        4.152060        0.000000
-------------------------------------------------------------------
Message Summary by Rank Pair:
SRank    DRank  Messages (minsize,maxsize)/[bin]    Totalbytes
-------------------------------------------------------------------
0
             1      1          (4, 4)                 4
                    1          [0..64]                4
-------------------------------------------------------------------
1
             0      1          (8, 8)                 8
                    1         [0..64]                 8
-------------------------------------------------------------------
```

Using the profiling interface

The MPI profiling interface provides a mechanism by which implementors of profiling tools can collect performance information without access to the underlying MPI implementation source code.

Because Platform MPI provides several options for profiling your applications, you might not need the profiling interface to write your routines. Platform MPI makes use of MPI profiling interface mechanisms to provide the diagnostic library for debugging. In addition, Platform MPI provides tracing and lightweight counter instrumentation.

The profiling interface allows you to intercept calls made by the user program to the MPI library. For example, you might want to measure the time spent in each call to a specific library routine or to create a log file. You can collect information of interest and then call the underlying MPI implementation through an alternate entry point as described below.

Routines in the Platform MPI library begin with the MPI_ prefix. Consistent with the Profiling Interface section of the MPI 1.2 standard, routines are also accessible using the PMPI_ prefix (for example, MPI_Send and PMPI_Send access the same routine).

To use the profiling interface, write wrapper versions of the MPI library routines you want the linker to intercept. These wrapper routines collect data for some statistic or perform some other action. The wrapper then calls the MPI library routine using the PMPI_ prefix.

Fortran profiling interface

When writing profiling routines, do not call Fortran entry points from C profiling routines, and visa versa. To profile Fortran routines, separate wrappers must be written.

For example:

```
#include <stdio.h>
#include <mpi.h>
int MPI_Send(void *buf, int count, MPI_Datatype type,
int to, int tag, MPI_Comm comm)
{
printf("Calling C MPI_Send to %d\n", to);
return PMPI_Send(buf, count, type, to, tag, comm);
}
#pragma weak (mpi_send mpi_send)
void mpi_send(void *buf, int *count, int *type, int *to,
int *tag, int *comm, int *ierr)
{
     printf("Calling Fortran MPI_Send to %d\n", *to);
pmpi_send(buf, count, type, to, tag, comm, ierr);
```

C++ profiling interface

The Platform MPI C++ bindings are wrappers to C calls. No profiling library exists for C++ bindings. To profile the C++ interface, write the equivalent C wrapper version of the MPI library routines you want to profile. For details on profiling the C MPI libraries, see the section above.

CHAPTER 6

Tuning

This chapter provides information about tuning Platform MPI applications to improve performance.

The tuning information in this chapter improves application performance in most but not all cases. Use this information together with the output from counter instrumentation to determine which tuning changes are appropriate to improve your application's performance.

When you develop Platform MPI applications, several factors can affect performance. These factors are outlined in this chapter.

Tunable parameters

Platform MPI provides a mix of command-line options and environment variables that can be used to influence the behavior and performance of the library. The options and variables of interest to performance tuning include the following:

MPI_FLAGS=y

This option can be used to control the behavior of the Platform MPI library when waiting for an event to occur, such as the arrival of a message.

MPI_TCP_CORECVLIMIT

Setting this variable to a larger value can allow Platform MPI to use more parallelism during its low-level message transfers, but it can greatly reduce performance by causing switch congestion.

MPI_SOCKBUFSIZE

Increasing this value has shown performance gains for some applications running on TCP networks.

-cpu_bind, MPI_BIND_MAP, MPI_CPU_AFFINITY, MPI_CPU_SPIN

The -cpu_bind command-line option and associated environment variables can improve the performance of many applications by binding a process to a specific CPU.

-intra

The -intra command-line option controls how messages are transferred to local processes and can impact performance when multiple ranks execute on a host.

MPI_RDMA_INTRALEN, MPI_RDMA_MSGSIZE, MPI_RDMA_NENVELOPE

These environment variables control aspects of the way message traffic is handled on RDMA networks. The default settings have been carefully selected for most applications. However, some applications might benefit from adjusting these values depending on their communication patterns. For more information, see the corresponding manpages.

MPI_USE_LIBELAN_SUB

Setting this environment variable may provide some performance benefits on the ELAN interconnect. However, some applications may experience resource problems.

Message latency and bandwidth

Latency is the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

Latency often depends on the length of messages being sent. An application's messaging behavior can vary greatly based on whether a large number of small messages or a few large messages are sent.

Message bandwidth is the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second. Bandwidth becomes important when message sizes are large.

To improve latency, bandwidth, or both:

- Reduce the number of process communications by designing applications that have coarse-grained parallelism.
- Use derived, contiguous data types for dense data structures to eliminate unnecessary byte-copy operations in some cases. Use derived data types instead of MPI_Pack and MPI_Unpack if possible. Platform MPI optimizes noncontiguous transfers of derived data types.
- Use collective operations when possible. This eliminates the overhead of using MPI_Send and MPI_Recv when one process communicates with others. Also, use the Platform MPI collectives rather than customizing your own.
- Specify the source process rank when possible when calling MPI routines. Using *MPI_ANY_SOURCE* can increase latency.
- Double-word align data buffers if possible. This improves byte-copy performance between sending and receiving processes because of double-word loads and stores.
- Use MPI_Recv_init and MPI_Startall instead of a loop of MPI_Irecv calls in cases where requests might not complete immediately. For example, suppose you write an application with the following code section:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Irecv(buf[i], count, dtype, i, 0, comm, &requests[j++]);
}
MPI_Waitall(size-1, requests, statuses);
```

Suppose that one of the iterations through MPI_Irecv does not complete before the next iteration of the loop. In this case, Platform MPI tries to progress both requests. This progression effort could continue to grow if succeeding iterations also do not complete immediately, resulting in a higher latency.

However, you could rewrite the code section as follows:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Recv_init(buf[i], count, dtype, i, 0, comm,
                                    &requests[j++]);
}
MPI_Startall(size-1, requests);
MPI_Waitall(size-1, requests, statuses);
```

In this case, all iterations through MPI_Recv_init are progressed just once when MPI_Startall is called. This approach avoids the additional progression overhead when using MPI_Irecv and can reduce application latency.

Multiple network interfaces

You can use multiple network interfaces for interhost communication while still having intrahost exchanges. In this case, the intrahost exchanges use shared memory between processes mapped to different same-host IP addresses.

To use multiple network interfaces, you must specify which MPI processes are associated with each IP address in your appfile.

For example, when you have two hosts, host 0 and host 1, each communicating using two Ethernet cards, ethernet 0 and ethernet 1, you have four host names as follows:

- host0-ethernet0
- host0-ethernet1
- host1-ethernet0
- host1-ethernet1

If your executable is called `work.exe` and uses 64 processes, your appfile should contain the following entries:

```
-h host0-ethernet0 -np 16 work.exe
-h host0-ethernet1 -np 16 work.exe
-h host1-ethernet0 -np 16 work.exe
-h host1-ethernet1 -np 16 work.exe
```

Now, when the appfile is run, 32 processes run on host 0 and 32 processes run on host 1.

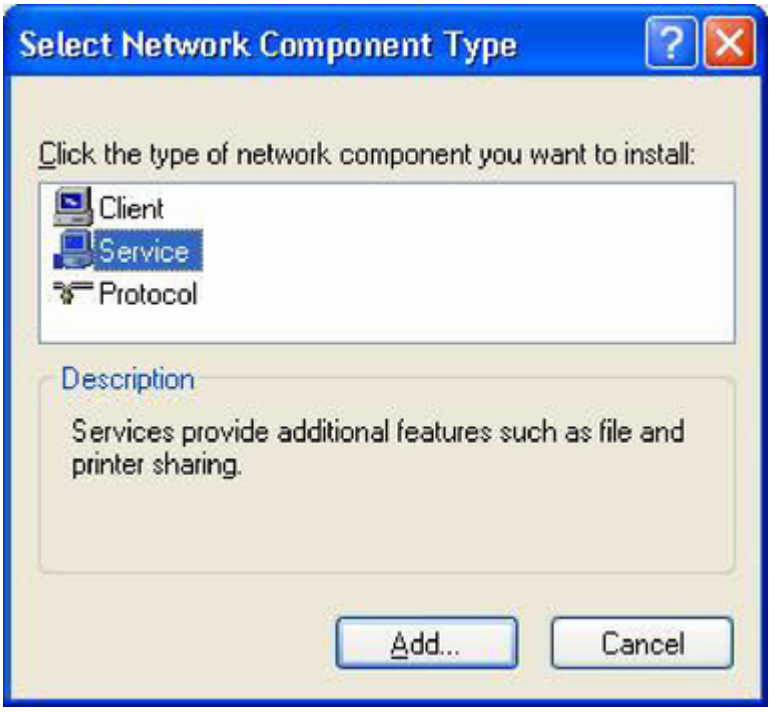


Figure 1: Multiple network interfaces

Host 0 processes with rank 0 - 15 communicate with processes with rank 16 - 31 through shared memory (shmem). Host 0 processes also communicate through the host 0-ethernet 0 and the host 0-ethernet 1 network interfaces with host 1 processes.

Processor subscription

Subscription refers to the match of processors and active processes on a host. The following table lists possible subscription types:

Table 18: Subscription types

Subscription type	Description
Under-subscribed	More processors than active processes
Fully subscribed	Equal number of processors and active processes
Over-subscribed	More active processes than processors

When a host is over-subscribed, application performance decreases because of increased context switching.

Context switching can degrade application performance by slowing the computation phase, increasing message latency, and lowering message bandwidth. Simulations that use timing-sensitive algorithms can produce unexpected or erroneous results when run on an over-subscribed system.

Processor locality

The `mpirun` option -cpu_bind binds a rank to a locality domain (ldom) to prevent a process from moving to a different ldom after start-up. The binding occurs before the MPI application is executed.

Similar results can be accomplished using mpsched but this has the advantage of being a more load-based distribution, and works well in psets and across multiple machines.

Binding ranks to ldoms (-cpu_bind)

On SMP systems, processes sometimes move to a different ldom shortly after start-up or during execution. This increases memory latency and can cause slower performance because the application is now accessing memory across cells.

Applications that are very memory-latency sensitive can show large performance degradation when memory access is mostly off-cell.

To solve this problem, ranks must reside in the same ldom where they were created. To accomplish this, Platform MPI provides the -cpu_bind flag, which locks down a rank to a specific ldom and prevents it from moving during execution. To accomplish this, the -cpu_bind flag preloads a shared library at start-up for each process, which does the following:

1. Spins for a short time in a tight loop to let the operating system distribute processes to CPUs evenly.
2. Determines the current CPU and ldom of the process. If no oversubscription occurs on the current CPU, it locks the process to the ldom of that CPU.

This evenly distributes the ranks to CPUs, and prevents the ranks from moving to a different ldom after the MPI application starts, preventing cross-memory access.

For more information, see refer to -cpu_bind in the `mpirun` documentation.

MPI routine selection

To achieve the lowest message latencies and highest message bandwidths for point-to-point synchronous communications, use the MPI blocking routines MPI_Send and MPI_Recv. For asynchronous communications, use the MPI nonblocking routines MPI_Isend and MPI_Irecv.

When using blocking routines, avoid pending requests. MPI must advance nonblocking messages, so calls to blocking receives must advance pending requests, occasionally resulting in lower application performance.

For tasks that require collective operations, use the relevant MPI collective routine. Platform MPI takes advantage of shared memory to perform efficient data movement and maximize your application's communication performance.

Multilevel parallelism

Consider the following to improve the performance of applications that use multilevel parallelism:

- Use the MPI library to provide coarse-grained parallelism and a parallelizing compiler to provide fine-grained (that is, thread-based) parallelism. A mix of coarse- and fine-grained parallelism provides better overall performance.
- Assign only one multithreaded process per host when placing application processes. This ensures that enough processors are available as different process threads become active.

Coding considerations

The following are suggestions and items to consider when coding your MPI applications to improve performance:

- Use Platform MPI collective routines instead of coding your own with point-to-point routines because Platform MPI's collective routines are optimized to use shared memory where possible for performance.
- Use commutative MPI reduction operations.
  - Use the MPI predefined reduction operations whenever possible because they are optimized.
  - When defining reduction operations, make them commutative. Commutative operations give MPI more options when ordering operations, allowing it to select an order that leads to best performance.
- Use MPI derived data types when you exchange several small size messages that have no dependencies.
- Minimize your use of MPI_Test() polling schemes to reduce polling overhead.
- Code your applications to avoid unnecessary synchronization. Avoid MPI_Barrier calls. Typically an application can be modified to achieve the same result using targeted synchronization instead of collective calls. For example, in many cases a token-passing ring can be used to achieve the same coordination as a loop of barrier calls.

Using HP Caliper

HP Caliper is a general-purpose performance analysis tool for applications, processes, and systems.

HP Caliper allows you to understand the performance and execution of an application, and identify ways to improve run-time performance.

Note:

When running Platform MPI applications under HP Caliper on Linux hosts, it might be necessary to set the *HPMPI_NOPROPAGATE_SUSP* environment variable to prevent application aborts.

% **setenv HPMPI_NOPROPAGATE_SUSP 1**

% **export HPMPI_NOPROPAGATE_SUSP=1**

CHAPTER 7

Debugging and Troubleshooting

This chapter describes debugging and troubleshooting Platform MPI applications.

Debugging Platform MPI applications

Platform MPI allows you to use single-process debuggers to debug applications. The available debuggers are ADB, DDE, XDB, WDB, GDB, and PATHDB. To access these debuggers, set options in the *MPI_FLAGS* environment variable. Platform MPI also supports the multithread multiprocess debugger, TotalView on Linux.

In addition to the use of debuggers, Platform MPI provides a diagnostic library (DLIB) for advanced error checking and debugging. Platform MPI also provides options to the environment variable *MPI_FLAGS* that report memory leaks (l), force MPI errors to be fatal (f), print the MPI job ID (j), and other functionality.

This section discusses single- and multi-process debuggers and the diagnostic library.

Using a single-process debugger

Because Platform MPI creates multiple processes and ADB, DDE, XDB, WDB, GDB, and PATHDB only handle single processes, Platform MPI starts one debugger session per process. Platform MPI creates processes in MPI_Init, and each process instantiates a debugger session. Each debugger session in turn attaches to the process that created it. Platform MPI provides *MPI_DEBUG_CONT* to control the point at which debugger attachment occurs. *MPI_DEBUG_CONT* is a variable that Platform MPI uses to temporarily halt debugger progress beyond MPI_Init. By default, *MPI_DEBUG_CONT* is set to 0 and you must reset it to 1 to allow the debug session to continue past MPI_Init.

Complete the following when you use a single-process debugger:

1. Set the eadb, exdb, edde, ewdb, egdb, or epathdb option in the *MPI_FLAGS* environment variable to use the ADB, XDB, DDE, WDB, GDB, or PATHDB debugger respectively.
2. On remote hosts, set *DISPLAY* to point to your console. In addition, use *xhost* to allow remote hosts to redirect their windows to your console.
3. Run your application.

   When your application enters MPI_Init, Platform MPI starts one debugger session per process and each debugger session attaches to its process.
4. (Optional) Set a breakpoint anywhere following MPI_Init in each session.
5. Set the global variable *MPI_DEBUG_CONT* to 1 using each session's command-line interface or graphical user interface. The syntax for setting the global variable depends upon which debugger you use:

   `(adb)` **mpi_debug_cont/w 1**

   `(dde)` **set mpi_debug_cont = 1**

   `(xdb)` **print *MPI_DEBUG_CONT = 1**

   `(wdb)` **set MPI_DEBUG_CONT = 1**

   `(gdb)` **set MPI_DEBUG_CONT = 1**
6. Issue the relevant debugger command in each session to continue program execution.

   Each process runs and stops at the breakpoint you set after MPI_Init.
7. Continue to debug each process using the relevant commands for your debugger.

Using a multiprocess debugger

Platform MPI supports the TotalView debugger on Linux. The preferred method when you run TotalView with Platform MPI applications is to use the `mpirun` run-time utility command.

For example,

% **$MPI_ROOT/bin/mpicc *myprogram.c* -g**

% **$MPI_ROOT/bin/mpirun -tv -np 2 a.out**

In this example, myprogram.c is compiled using the Platform MPI compiler utility for C programs. The executable file is compiled with source line information and then `mpirun` runs the `a.out` MPI program:

-g

Specifies that the compiler generate the additional information needed by the symbolic debugger.

-np 2

Specifies the number of processes to run (2, in this case).

-tv

Specifies that the MPI ranks are run under TotalView.

Alternatively, use `mpirun` to invoke an appfile:

% **$MPI_ROOT/bin/mpirun -tv -f *my_appfile***

-tv

Specifies that the MPI ranks are run under TotalView.

-f appfile

Specifies that `mpirun` parses *appfile* to get program and process count information for the run.

By default, `mpirun` searches for TotalView in your *PATH*. You can also define the absolute path to TotalView using the *TOTALVIEW* environment variable:

% **setenv TOTALVIEW=*/opt/totalview/bin/totalview* [*totalview-options*]**

The *TOTALVIEW* environment variable is used by `mpirun`.

Note:

When attaching to a running MPI application that was started using appfiles, attach to the MPI daemon process to enable debugging of all the MPI ranks in the application. You can identify the daemon process as the one at the top of a hierarchy of MPI jobs (the daemon also usually has the lowest PID among the MPI jobs).

Limitations

The following limitations apply to using TotalView with Platform MPI applications:

- All executable files in your multihost MPI application must reside on your local machine, that is, the machine on which you start TotalView.

TotalView multihost example

The following example demonstrates how to debug a typical Platform MPI multihost application using TotalView, including requirements for directory structure and file locations.

The MPI application is represented by an appfile, named `my_appfile`, which contains the following two lines:

```
-h local_host -np 2 /path/to/program1 -h remote_host -np 2 /path/to/program2
```

`my_appfile` resides on the local machine (local_host) in the `/work/mpiapps/total` directory.

To debug this application using TotalView do the following. In this example, TotalView is invoked from the local machine.

1. Place your binary files in accessible locations.
   - `/path/to/program1` exists on local_host
   - `/path/to/program2` exists on remote_host

     To run the application under TotalView, the directory layout on your local machine, with regard to the MPI executable files, must mirror the directory layout on each remote machine. Therefore, in this case, your setup must meet the following additional requirement:
   - `/path/to/program2` exists on local_host
2. In the `/work/mpiapps/total` directory on local_host, invoke TotalView by passing the -tv option to `mpirun`:

   % **$MPI_ROOT/bin/mpirun -tv -f *my_appfile***

Using the diagnostics library

Platform MPI provides a diagnostics library (DLIB) for advanced run-time error checking and analysis. DLIB provides the following checks:

- Message signature analysis : Detects type mismatches in MPI calls. For example, in the two calls below, the send operation sends an integer, but the matching receive operation receives a floating-point number.

  ```
  if (rank == 1) then
  MPI_Send(&buf1, 1, MPI_INT, 2, 17, MPI_COMM_WORLD);
  else if (rank == 2)
  MPI_Recv(&buf2, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD, &status);
  ```
- MPI object-space corruption : Detects attempts to write into objects such as MPI_Comm, MPI_Datatype, MPI_Request, MPI_Group, and MPI_Errhandler.
- Multiple buffer writes : Detects whether the data type specified in a receive or gather operation causes MPI to write to a user buffer more than once.

To disable these checks or enable formatted or unformatted printing of message data to a file, set the *MPI_DLIB_FLAGS* environment variable options appropriately.

To use the diagnostics library, specify the -ldmpi option to the build scripts when you compile your application. This option is supported on Linux and Windows.

Note:

Using DLIB reduces application performance. Also, you cannot use DLIB with instrumentation.

Enhanced debugging output

Platform MPI provides the stdio option to allow improved readability and usefulness of MPI processes stdout and stderr. Options have been added for handling standard input:

- Directed: Input is directed to a specific MPI process.
- Broadcast: Input is copied to the stdin of all processes.
- Ignore: Input is ignored.

The default behavior when using stdio is to ignore standard input.

Additional options are available to avoid confusing interleaving of output:

- Line buffering, block buffering, or no buffering
- Prepending of processes ranks to stdout and stderr
- Simplification of redundant output

This functionality is not provided when using -srun or -prun. Refer to the --label option of `srun` for similar functionality.

Backtrace functionality

Platform MPI handles several common termination signals on PA-RISC differently than earlier versions of Platform MPI. If any of the following signals are generated by an MPI application, a stack trace is printed prior to termination:

- `SIGBUS` : bus error
- `SIGSEGV` : segmentation violation
- `SIGILL` : illegal instruction
- `SIGSYS` : illegal argument to system call

The backtrace is helpful in determining where the signal was generated and the call stack at the time of the error. If a signal handler is established by the user code before calling MPI_Init, no backtrace is printed for that signal type and the user's handler is solely responsible for handling the signal. Any signal handler installed after MPI_Init also overrides the backtrace functionality for that signal after the point it is established. If multiple processes cause a signal, each of them prints a backtrace.

In some cases, the prepending and buffering options available in Platform MPI standard IO processing are useful in providing more readable output of the backtrace information.

The default behavior is to print a stack trace.

Backtracing can be turned off entirely by setting the environment variable *MPI_NOBACKTRACE*.

Debugging tutorial for Windows

A browser-based tutorial is provided that contains information on how to debug applications that use Platform MPI in the Windows environment. The tutorial provides step-by-step procedures for performing common debugging tasks using Visual Studio 2005.

The tutorial is located in the `%MPI_ROOT%\help` subdirectory.

Troubleshooting Platform MPI applications

This section describes limitations in Platform MPI, common difficulties, and hints to help you overcome those difficulties and get the best performance from your Platform MPI applications. Check this information first when you troubleshoot problems. The topics covered are organized by development task and also include answers to frequently asked questions:

To get information about the version of Platform MPI installed, use the `mpirun -version` command. The following is an example of the command and its output:

% **mpirun -version $MPI_ROOT/bin/mpicc**

```
$MPI_ROOT/bin/mpicc:
Platform MPI 02.01.01.00 (dd/mm/yyyy) B6060BA
```

This command returns the Platform MPI version number, the release date, Platform MPI product numbers, and the operating system version.

For Linux systems, use

% **ident $MPI_ROOT/bin/mpirun**

or

% **rpm -qa | grep pcmpi**

For Windows systems, use

> **"%MPI_ROOT%\bin\mprun" -version**

```
mpirun: Platform MPI 01.00.00.00 Windows 32 major version 100 minor version 0
```

Building on Linux

You can solve most build-time problems by referring to the documentation for the compiler you are using.

If you use your own build script, specify all necessary input libraries. To determine what libraries are needed, check the contents of the compilation utilities stored in the Platform MPI `$MPI_ROOT/bin` subdirectory.

Platform MPI supports a 64-bit version of the MPI library on 64-bit platforms. Both 32-bit and 64-bit versions of the library are shipped on 64-bit platforms. You cannot mix 32-bit and 64-bit executables in the same application.

Platform MPI does not support Fortran applications that are compiled with the following option:

+autodblpad : Fortran 77 programs

Building on Windows

Make sure you are running the build wrappers (i.e., `mpicc`, `mpif90`) in a compiler command window. This window is usually an option on the Start > All Programs menu. Each compiler vendor provides a command window option that includes all necessary paths for compiler and libraries.

On Windows, the Platform MPI libraries include the bitness in the library name. Platform MPI provides support for 32-bit and 64-bit libraries. The .lib files are located in `%MPI_ROOT%\lib`.

Starting on Linux

When starting multihost applications using an appfile, make sure that:

- All remote hosts are listed in your .rhosts file on each machine and you can `remsh` to the remote machines. The `mpirun` command has the -ck option, which you can use to determine whether the hosts and programs specified in your MPI application are available, and whether there are access or permission problems. MPI `remsh` can be used to specify other commands to be used, such as `ssh`, instead of `remsh`.
- Application binaries are available on the necessary remote hosts and are executable on those machines.
- The -sp option is passed to `mpirun` to set the target shell *PATH* environment variable. You can set this option in your appfile.
- The .cshrc file does not contain `tty` commands such as `stty` if you are using a `/bin/csh`-based shell.

Starting on Windows

When starting multihost applications using Windows HPCS:

- Don't forget the -ccp flag.
- Use UNC paths for your file names. Drives are usually not mapped on remote nodes.
- If using the AutoSubmit feature, make sure you are running from a mapped network drive and don't specify file paths for binaries. Platform MPI converts the mapped drive to a UNC path and set *MPI_WORKDIR* to your current directory. If you are running on a local drive, Platform MPI cannot map this to a UNC path.
- Don't submit scripts or commands that require a command window. These commands usually fail when trying to 'change directory' to a UNC path.
- Don't forget to use quotation marks for file names or commands with paths that have spaces. The default Platform MPI installation location includes spaces:

  ```
  "C:\Program Files (x86)\Platform-MPI\bin\mpirun"
  ```

  or

  ```
  "%MPI_ROOT%\bin\mpirun"
  ```

When starting multihost applications using appfiles on Windows 2003/XP, verify the following:

- Platform MPI Remote Launch service is registered and started on all remote nodes. Check this by accessing the list of Windows services through Administrator Tools > Services. Look for the 'Platform MPI Remote Launch' service.
- Platform MPI is installed in the same location on all remote nodes. All Platform MPI libraries and binaries must be in the same *MPI_ROOT*.
- Application binaries are accessible from remote nodes. If the binaries are located on a file share, use the UNC path (i.e., `\\node\share\path`) to refer to the binary, because these might not be properly mapped to a drive letter by the authenticated logon token.
- If a password is not already cached, use the -cache option for your first run, or use the -pass option on all runs so the remote service can authenticate with network resources. Without these options (or using -nopass), remote processes cannot access network shares.
- If problems occur when trying to launch remote processes, use the `mpidiag` tool to verify remote authentication and access. Also view the event logs to see if the service is issuing errors.
- Don't forget to use quotation marks for file names or commands with paths that have spaces. The default Platform MPI installation location includes spaces:

  ```
  "C:\Program Files (x86)\Platform-MPI\bin\mpirun"
  ```

  or

  ```
  "%MPI_ROOT%\bin\mpirun"
  ```

Running on Linux and Windows

Run-time problems originate from many sources and may include the following:

Shared memory

When an MPI application starts, each MPI daemon attempts to allocate a section of shared memory. This allocation can fail if the system-imposed limit on the maximum number of allowed shared-memory identifiers is exceeded or if the amount of available physical memory is not sufficient to fill the request. After shared-memory allocation is done, every MPI process attempts to attach to the shared-memory region of every other process residing on the same host. This shared memory allocation can fail if the system is not configured with enough available shared memory. Consult with your system administrator to change system settings. Also, *MPI_GLOBMEMSIZE* is available to control how much shared memory Platform MPI tries to allocate.

Message buffering

According to the MPI standard, message buffering may or may not occur when processes communicate with each other using MPI_Send. MPI_Send buffering is at the discretion of the MPI implementation. Therefore, take care when coding communications that depend upon buffering to work correctly.

For example, when two processes use MPI_Send to simultaneously send a message to each other and use MPI_Recv to receive messages, the results are unpredictable. If the messages are buffered, communication works correctly. However, if the messages are not buffered, each process hangs in MPI_Send waiting for MPI_Recv to take the message. For example, a sequence of operations (labeled "Deadlock") as illustrated in the following table would result in such a deadlock. This table also illustrates the sequence of operations that would avoid code deadlock:

Table 19: Non-buffered messages and deadlock

Deadlock		No Deadlock	
Process 1	**Process 2**	**Process 1**	**Process 2**
MPI_Send(,...2,....)	MPI_Send(,...1,....)	MPI_Send(,...2,....)	MPI_Recv(,...1,....)
MPI_Recv(,...2,....)	MPI_Recv(,...1,....)	MPI_Recv(,...2,....)	MPI_Send(,...1,....)

Propagation of environment variables

When working with applications that run on multiple hosts using an appfile, if you want an environment variable to be visible by all application ranks you must use the -e option with an appfile or as an argument to `mpirun`.

One way to accomplish this is to set the -e option in the appfile:

-h *remote_host* -e *var=val* [-np#] *program* [*args*]

On HP XC, systems the environment variables are automatically propagated by `srun`. Environment variables are established with `setenv` or `export` and passed to MPI processes by the SLURM `srun` utility. Thus, on HP XC systems, it is not necessary to use the "-e name=value" approach to passing environment variables. Although the "-e name=value" also works on HP XC systems using SLURM's `srun`.

Fortran 90 programming features

The MPI 1.1 standard defines bindings for Fortran 77 but not Fortran 90.

Although most Fortran 90 MPI applications work using the Fortran 77 MPI bindings, some Fortran 90 features can cause unexpected behavior when used with Platform MPI.

In Fortran 90, an array is not always stored in contiguous memory. When noncontiguous array data is passed to a Platform MPI subroutine, Fortran 90 copies the data into temporary storage, passes it to the Platform MPI subroutine, and copies it back when the subroutine returns. As a result, Platform MPI is given the address of the copy but not of the original data.

In some cases, this copy-in and copy-out operation can cause a problem. For a nonblocking Platform MPI call, the subroutine returns immediately and the temporary storage is deallocated. When Platform MPI tries to access the already invalid memory, the behavior is unknown. Moreover, Platform MPI operates close to the system level and must know the address of the original data. However, even if the address is known, Platform MPI does not know if the data is contiguous or not.

UNIX open file descriptors

UNIX imposes a limit to the number of file descriptors that application processes can have open at one time. When running a multihost application, each local process opens a socket to each remote process. A Platform MPI application with a large amount of off-host processes can quickly reach the file descriptor limit. Ask your system administrator to increase the limit if your applications frequently exceed the maximum.

External input and output

You can use stdin, stdout, and stderr in applications to read and write data. By default, Platform MPI does not perform processing on stdin or stdout. The controlling tty determines stdio behavior in this case.

This functionality is not provided when using -srun or -prun.

If your application depends on the `mpirun` option -stdio=I to broadcast input to all ranks, and you are using SLURM's `srun` on an HP XC system, then a reasonable substitute is --stdin=all. For example:

```
% mpirun -srun --stdin-all ...
```

For similar functionality, refer to the --label option of `srun`.

Platform MPI does provide optional stdio processing features. stdin can be targeted to a specific process, or can be broadcast to every process. stdout processing includes buffer control, prepending MPI rank numbers, and combining repeated output.

Platform MPI standard IO options can be set by using the following options to `mpirun`:

mpirun -stdio=[bline[*#>0*] | bnone[*#>0*] | b[*#>0*], [p], [r[*#>1*]], [i[*#*]], files, none

where

i

Broadcasts standard input to all MPI processes.

i[*#*]

Directs standard input to the process with the global rank #.

The following modes are available for buffering:

b[*#>0*]

Specifies that the output of a single MPI process is placed to the standard out of `mpirun` after # bytes of output have been accumulated.

bnone[*#>0*]

The same as b[#] except that the buffer is flushed when it is full and when it is found to contain data. Essentially provides no buffering from the user's perspective.

bline[*#>0*]

Displays the output of a process after a line feed is encountered, or if the # byte buffer is full.

The default value of # in all cases is 10 KB

The following option is available for prepending:

p

Enables prepending. The global rank of the originating process is prepended to stdout and stderr output. Although this mode can be combined with any buffering mode, prepending makes the most sense with the modes b and bline.

The following option is available for combining repeated output:

r[*#>1*]

Combines repeated identical output from the same process by prepending a multiplier to the beginning of the output. At most, # maximum repeated outputs are accumulated without display. This option is used only with bline. The default value of # is infinity.

The following options are available for using file settings:

files

Specifies that the standard input, output and error of each rank is to be taken from the files specified by the environment variables *MPI_STDIO_INFILE*, *MPI_STDIO_OUTFILE* and *MPI_STDIO_ERRFILE*. If these environment variables are not set, `/dev/null` or `NUL` is used. In addition, these file specifications can include the substrings `%%`, `%h`, `%p`, and `%r`, which are expanded to `%`, `hostname`, `process id`, and `rank number` in *MPI_COMM_WORLD*. The files option causes the stdio options p, r, and I to be ignored.

none

This option is equivalent to setting -stdio=files with *MPI_STDIO_INFILE*, *MPI_STDIO_OUTFILE* and *MPI_STDIO_ERRFILE* all set to `/dev/null` or `NUL`.

Completing

In Platform MPI, MPI_Finalize is a barrier-like collective routine that waits until all application processes have called it before returning. If your application exits without calling MPI_Finalize, pending requests might not complete.

When running an application, `mpirun` waits until all processes have exited. If an application detects an MPI error that leads to program termination, it calls MPI_Abort instead.

You might want to code your error conditions using MPI_Abort, which cleans up the application.

Each Platform MPI application is identified by a job ID, unique on the server where `mpirun` is invoked. If you use the -j option, `mpirun` prints the job ID of the application that it runs. Then you can invoke `mpijob` with the job ID to display the status of your application.

If your application hangs or terminates abnormally, you can use `mpiclean` to kill lingering processes and shared-memory segments. `mpiclean` uses the job ID from `mpirun -j` to specify the application to terminate.

Testing the network on Linux

Often, clusters might have Ethernet and some form of higher speed interconnect such as InfiniBand. This section describes how to use the ping_pong_ring.c example program to confirm that you can run using the desired interconnect.

Running a test like this, especially on a new cluster, is useful to ensure that the relevant network drivers are installed and that the network hardware is functioning. If any machine has defective network cards or cables, this test can also be useful at identifying which machine has the problem.

To compile the program, set the *MPI_ROOT* environment variable (not required, but recommended) to a value such as `/opt/platform_mpi` (for Linux) and then run

% **export MPI_CC=gcc** (using whatever compiler you want)

% **$MPI_ROOT/bin/mpicc -o pp.x $MPI_ROOT/help/ping_pong_ring.c**

Although `mpicc` performs a search for the compiler to use if you don't specify *MPI_CC*, it is preferable to be explicit.

If you have a shared file system, it is easiest to put the resulting `pp.x` executable there, otherwise you must explicitly copy it to each machine in your cluster.

Use the start-up relevant for your cluster. Your situation should resemble one of the following:

- If no job scheduler (such as `srun`, `prun`, or LSF) is available, run a command like this:

  $MPI_ROOT/bin/mpirun -prot -hostlist hostA,hostB,...hostZ pp.x

  You might need to specify the remote shell command to use (the default is ssh) by setting the *MPI_REMSH* environment variable. For example:

  % **export MPI_REMSH="rsh -x"** (optional)
- If LSF is being used, create an appfile such as this:

  ```
  -h hostA -np 1 /path/to/pp.x
  -h hostB -np 1 /path/to/pp.x
  -h hostC -np 1 /path/to/pp.x
  ...
  -h hostZ -np 1 /path/to/pp.x
  ```

  Then run one of the following commands:

  % **bsub pam -mpi $MPI_ROOT/bin/mpirun -prot -f appfile**

  % **bsub pam -mpi $MPI_ROOT/bin/mpirun -prot -f appfile -- 1000000**

  When using LSF, the host names in the appfile are ignored.
- If the `srun` command is available, run a command like this:

  % **$MPI_ROOT/bin/mpirun -prot -srun -N 8 -n 8 path/to/pp.x**

  % **$MPI_ROOT/bin/mpirun -prot -srun -N 8 -n 8 path/to/pp.x 1000000**

  Replacing "8" with the number of hosts.

  Or if LSF is being used, then the command to run might be this:

  % **bsub -I -n 16 $MPI_ROOT/bin/mpirun -prot -srun /path/to/pp.x**

```
% bsub -I -n 16 $MPI_ROOT/bin/mpirun -prot -srun /path/to/pp.x 1000000
```

- If the `prun` command is available, use the same commands as above for `srun`, replacing `srun` with `prun`.

In each case above, the first `mpirun` command uses 0 bytes per message and verifies latency. The second `mpirun` command uses 1000000 bytes per message and verifies bandwidth.

Example output might look like:

```
Host 0 -- ip 192.168.9.10 -- ranks 0
Host 1 -- ip 192.168.9.11 -- ranks 1
Host 2 -- ip 192.168.9.12 -- ranks 2
Host 3 -- ip 192.168.9.13 -- ranks 3

host  | 0    1    2    3
======|====================
    0 : SHM  VAPI VAPI VAPI
    1 : VAPI SHM  VAPI VAPI
    2 : VAPI VAPI SHM  VAPI
    3 : VAPI VAPI VAPI SHM

[0:hostA] ping-pong 0 bytes ...
0 bytes: 4.24 usec/msg
[1:hostB] ping-pong 0 bytes ...
0 bytes: 4.26 usec/msg
[2:hostC] ping-pong 0 bytes ...
0 bytes: 4.26 usec/msg
[3:hostD] ping-pong 0 bytes ...
0 bytes: 4.24 usec/msg
```

The table showing SHM/VAPI is printed because of the -prot option (print protocol) specified in the `mpirun` command. It could show any of the following settings:

- `VAPI`: VAPI on InfiniBand
- `UDAPL`: uDAPL on InfiniBand
- `IBV`: IBV on InfiniBand
- `PSM`: PSM on InfiniBand
- `MX`: Myrinet MX
- `IBAL`: on InfiniBand (for Windows only)
- `IT`: IT-API on InfiniBand
- `GM`: Myrinet GM2
- `ELAN`: Quadrics Elan4
- `TCP`: TCP/IP
- `MPID`: daemon communication mode
- `SHM`: shared memory (intra host only)

If the table shows TCP for hosts, the host might not have correct network drivers installed.

If a host shows considerably worse performance than another, it can often indicate a bad card or cable.

Other possible reasons for failure could be:

- A connection on the switch is running in 1X mode instead of 4X mode.
- A switch has degraded a port to SDR (assumes DDR switch, cards).
- A degraded SDR port could be due to using a non-DDR cable.

If the run aborts with an error message, Platform MPI might have incorrectly determined what interconnect was available. One common way to encounter this problem is to run a 32-bit application on a 64-bit machine like an Opteron or Intel64. It's not uncommon for some network vendors to provide only 64-bit libraries.

Platform MPI determines which interconnect to use before it knows the application's bitness. To have proper network selection in that case, specify if the application is 32-bit when running on Opteron/Intel64 machines.

% **$MPI_ROOT/bin/mpirun -mpi32 ...**

Testing the network on Windows

Often, clusters might have Ethernet and some form of higher-speed interconnect such as InfiniBand. This section describes how to use the ping_pong_ring.c example program to confirm that you can run using the desired interconnect.

Running a test like this, especially on a new cluster, is useful to ensure that relevant network drivers are installed and that the network hardware is functioning. If any machine has defective network cards or cables, this test can also be useful for identifying which machine has the problem.

To compile the program, set the *MPI_ROOT* environment variable to the location of Platform MPI. The default is `"C:\Program Files (x86)\Platform Computing\Platform MPI"` for 64-bit systems, and `"C:\Program Files \Platform Computing\Platform MPI"` for 32-bit systems. This may already be set by the Platform MPI installation.

Open a command window for the compiler you plan on using. This includes all libraries and compilers in the path, and compiles the program using the `mpicc` wrappers:

"%MPI_ROOT%\bin\mpicc" -mpi64 out:pp.exe %MPI_ROOT%\help\ping_ping_ring.c"

Use the start-up for your cluster. Your situation should resemble one of the following:

- If running on Windows HPCS using appfile mode:

  Create an appfile such as this:

  -h hostA -np 1 \\node\share\path\to\pp.x

  -h hostB -np 1 \\node\share\path\to\pp.x

  -h hostC -np 1 \\node\share\path\to\pp.x

  Submit the command to the scheduler using Automatic scheduling from a mapped share drive:

  > "%MPI_ROOT%\bin\mpirun" -ccp -prot -f appfile

  > "%MPI_ROOT%\bin\mpirun" -ccp -prot -f appfile -- 1000000
- If running on Windows HPCS using automatic scheduling:

  Submit the command to the scheduler, but include the total number of processes needed on the nodes as the -np command. This is *not* the rank count when used in this fashion. Also include the -nodexflag to indicate only one rank/node.

  Assuming 4 CPUs/nodes in this cluster, the command would be:

  > "%MPI_ROOT%\bin\mpirun" -ccp -np 12 -nodex -prot ping_ping_ring.exe

  > "%MPI_ROOT%\bin\mpirun" -ccp -np 12 -nodex -prot ping_ping_ring.exe 1000000
- If running on Windows 2003/XP using appfile mode:

  Create an appfile such as this:

  -h hostA -np 1 \\node\share\path\to\pp.x

  -h hostB -np 1 \\node\share\path\to\pp.x

  -h hostC -np 1 \\node\share\path\to\pp.x

Submit the command to the scheduler using Automatic scheduling from a mapped share drive:

> **"%MPI_ROOT%\bin\mpirun" -ccp -prot -f appfile**

> **"%MPI_ROOT%\bin\mpirun" -ccp -prot -f appfile -- 1000000**

In each case above, the first `mpirun` command uses 0 bytes per message and verifies latency. The second `mpirun` command uses 1000000 bytes per messageand verifies bandwidth.

Example output might look like:

```
Host 0 -- ip 172.16.159.3 -- ranks 0
Host 1 -- ip 172.16.150.23 -- ranks 1
Host 2 -- ip 172.16.150.24 -- ranks 2
host | 0    1    2
=====|================
   0 : SHM  IBAL IBAL
   1 : IBAL SHM  IBAL
   2 : IBAL IBAL SHM
[0:mpiccp3] ping-pong 1000000 bytes ...
1000000 bytes: 1089.29 usec/msg
1000000 bytes: 918.03 MB/sec
[1:mpiccp4] ping-pong 1000000 bytes ...
1000000 bytes: 1091.99 usec/msg
1000000 bytes: 915.76 MB/sec
[2:mpiccp5] ping-pong 1000000 bytes ...
1000000 bytes: 1084.63 usec/msg
1000000 bytes: 921.97 MB/sec
```

The table showing SHM/IBAL is printed because of the -prot option (print protocol) specified in the `mpirun` command.

It could show any of the following settings:

- `IBAL`: IBAL on InfiniBand
- `MX`: Myrinet Express
- `TCP`: TCP/IP

  `MPID`: daemon communication mode
- `SHM`: shared memory (intra host only)

If one or more hosts show considerably worse performance than another, it can often indicate a bad card or cable.

If the run aborts with some kind of error message, it is possible that Platform MPI incorrectly determined which interconnect was available.

APPENDIX

A

Example Applications

This appendix provides example applications that supplement the conceptual information in this book about MPI in general and Platform MPI in particular. The example codes are also included in the `$MPI_ROOT/help` subdirectory of your Platform MPI product.

Table 20: Example applications shipped with Platform MPI

Name	Language	Description	-np Argument
send_receive.f	Fortran 77	Illustrates a simple send and receive operation.	-np >= 2
ping_pong.c	C	Measures the time it takes to send and receive data between two processes.	-np = 2
ping_pong_ring.c		Confirms that an application can run using the specified interconnect.	-np >= 2
compute_pi.f	Fortran 77	Computes pi by integrating f(x) =4/(1+x*x).	-np >= 1
master_worker.f90	Fortran 90	Distributes sections of an array and does computation on all sections in parallel.	-np >= 2
cart.C	C++	Generates a virtual topology.	-np = 4
communicator.c	C	Copies the default communicator *MPI_COMM_WORLD*.	-np = 2
multi_par.f	Fortran 77	Uses the alternating direction iterative (ADI) method on a two-dimensional compute region.	-np >= 1

Name	Language	Description	-np Argument
io.c	C	Writes data for each process to a separate file called iodata*x*, where*x* represents each process rank in turn. Then the data in iodata*x* is read back.	-np >= 1
thread_safe.c	C	Tracks the number of client requests handled and prints a log of the requests to stdout.	-np >= 2
sort.C	C++	Generates an array of random integers and sorts it.	-np >= 1
compute_pi_spawn.f	Fortran 77	A single initial rank spawns 3 new ranks that all perform the same computation as in `compute_pi.f`.	-np >= 1
ping_pong_clustertest.c	C	Identifies slower than average links in your high-speed interconnect.	-np >2
hello_world.c	C	Prints host name and rank.	-np >=1

These examples and the makefile are located in the `$MPI_ROOT/help` subdirectory. The examples are presented for illustration purposes only. They might not necessarily represent the most efficient way to solve a problem.

To build and run the examples, use the following procedure:

1. Change to a writable directory.
2. Copy all files from the help directory to the current writable directory:

   % **cp $MPI_ROOT/help/* .**
3. Compile all examples or a single example.

   To compile and run all examples in the `/help` directory, at the prompt enter:

   % **make**

   To compile and run the thread_safe.c program only, at the prompt enter:

   % **make thread_safe**

send_receive.f

In this Fortran 77 example, process 0 sends an array to other processes in the default communicator *MPI_COMM_WORLD*.

```
program maininclude 'mpif.h'
integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
double precision data(100)
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

```
if (size .eq. 1) then
print *, 'must have at least 2 processes'
call MPI_Finalize(ierr)
stop
endif
print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
if (rank .eq. src) then
to = dest
count = 10
tag = 2001
do i=1, 10
data(i) = 1
enddo
call MPI_Send(data, count, MPI_DOUBLE_PRECISION,
+                     to, tag, MPI_COMM_WORLD, ierr)
endif
if (rank .eq. dest) then
tag = MPI_ANY_TAG
count = 10
from = MPI_ANY_SOURCE
call MPI_Recv(data, count, MPI_DOUBLE_PRECISION,
+                     from, tag, MPI_COMM_WORLD, status, ierr)
call MPI_Get_Count(status, MPI_DOUBLE_PRECISION,
+                           st_count, ierr)
st_source = status(MPI_SOURCE)
st_tag = status(MPI_TAG)
print *, 'Status info: source = ', st_source,
+                 ' tag = ', st_tag, ' count = ', st_count
print *, rank, ' received', (data(i),i=1,10)
endif
call MPI_Finalize(ierr)
stop
end
```

send_receive output

The output from running the send_receive executable is shown below. The application was run with -np=10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  2 of  10 is alive
Process  3 of  10 is alive
Process  4 of  10 is alive
Process  5 of  10 is alive
Process  6 of  10 is alive
Process  7 of  10 is alive
Process  8 of  10 is alive
Process  9 of  10 is alive
Status info: source =  0 tag =  2001 count =  10
9 received 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
```

ping_pong.c

This C example is used as a performance benchmark tomeasure the amount of time it takes to send and receive data betweentwo processes. The buffers are aligned and offset from each other toavoid cache conflicts caused by direct process-to-process byte-copyoperations

To run this example:

1. Define the CHECK macro to check data integrity.
2. Increase the number of bytes to at least twice the cache size to obtain representative bandwidth measurements.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <mpi.h>
#define NLOOPS          1000
#define ALIGN           4096
main(argc, argv)
int                     argc;
char                    *argv[];
{
int             i, j;
double          start, stop;
int             nbytes = 0;
int             rank, size;
MPI_Status      status;
char            *buf;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size != 2) {
if ( ! rank) printf("ping_pong: must have two processes\n");
MPI_Finalize();
exit(0);
}
nbytes = (argc > 1) ? atoi(argv[1]) : 0;
if (nbytes < 0) nbytes = 0;
/*
* Page-align buffers and displace them in the cache to avoid collisions.
*/
buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
if (buf == 0) {
MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
exit(1);
}
buf = (char *) ((((unsigned long) buf) + (ALIGN - 1)) & ~(ALIGN - 1));
if (rank == 1) buf += 524288;
memset(buf, 0, nbytes);
/*
* Ping-pong.
*/
if (rank == 0) {
printf("ping-pong %d bytes ...\n", nbytes);
/*
* warm-up loop
*/
for (i = 0; i < 5; i++) {
MPI_Send(buf, nbytes, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
MPI_Recv(buf, nbytes, MPI_CHAR,1, 1,
MPI_COMM_WORLD, &status);
}
/*
* timing loop
*/
start = MPI_Wtime();
for (i = 0; i < NLOOPS; i++) {
#ifdef CHECK
for (j = 0; j < nbytes; j++) buf[j] = (char) (j + i);
#endif
MPI_Send(buf, nbytes, MPI_CHAR,1, 1000 + i, MPI_COMM_WORLD);
#ifdef CHECK
memset(buf, 0, nbytes);
#endif
MPI_Recv(buf, nbytes, MPI_CHAR,1, 2000 + i,
MPI_COMM_WORLD,&status);
#ifdef CHECK
for (j = 0; j < nbytes; j++) {
if (buf[j] != (char) (j + i)) {
printf("error: buf[%d] = %d, not %d\n",j,
buf[j], j + i);
break;
}
}
#endif
}
stop = MPI_Wtime();
```

```
printf("%d bytes: %.2f usec/msg\n",
nbytes, (stop - start) / NLOOPS / 2 * 1000000);
if (nbytes > 0) {
printf("%d bytes: %.2f MB/sec\n", nbytes,nbytes / 1000000./
((stop - start) / NLOOPS / 2));
}
}
else {
/*
* warm-up loop
*/
for (i = 0; i < 5; i++) {
MPI_Recv(buf, nbytes, MPI_CHAR,0, 1, MPI_COMM_WORLD, &status);
MPI_Send(buf, nbytes, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
}
for (i = 0; i < NLOOPS; i++) {
MPI_Recv(buf, nbytes, MPI_CHAR,0, 1000 + i,
MPI_COMM_WORLD,&status);
MPI_Send(buf, nbytes, MPI_CHAR,0, 2000 + i, MPI_COMM_WORLD);
}
}
MPI_Finalize();
exit(0);
}
```

ping_pong output

The output from running the ping_pong executable is shown below. The application was run with -np2.

```
ping-pong 0 bytes ...
0 bytes: 1.03 usec/msg
```

ping_pong_ring.c (Linux)

Often a cluster might have regular Ethernet and some form of higher-speed interconnect such as InfiniBand. This section describes how to use the ping_pong_ring.c example program to confirm that you can run using the desired interconnect.

Running a test like this, especially on a new cluster, is useful to ensure that the relevant network drivers are installed and that the network hardware is functioning. If any machine has defective network cards or cables, this test can also be useful to identify which machine has the problem.

To compile the program, set the *MPI_ROOT* environment variable (not required, but recommended) to a value such as `/opt/platform_mpi` (Linux) and then run:

`%` **export MPI_CC=gcc** (whatever compiler you want)

`%` **$MPI_ROOT/bin/mpicc -o pp.x $MPI_ROOT/help/ping_pong_ring.c**

Although `mpicc` will perform a search for what compiler to use if you don't specify *MPI_CC*, it is preferable to be explicit.

If you have a shared filesystem, it is easiest to put the resulting `pp.x` executable there, otherwise you must explicitly copy it to each machine in your cluster.

There are a variety of supported start-up methods, and you must know which is relevant for your cluster. Your situation should resemble one of the following:

1. No `srun`, `prun`, or HPCS job scheduler command is available.

   For this case you can create an appfile with the following:

   ```
   -h hostA -np 1 /path/to/pp.x
   -h hostB -np 1 /path/to/pp.x
   -h hostC -np 1 /path/to/pp.x
   ```

```
...
-h hostZ -np 1 /path/to/pp.x
```

And you can specify what remote shell command to use (Linux default is ssh) in the *MPI_REMSH* environment variable.

For example you might use:

% **export MPI_REMSH="rsh -x"**(optional)

Then run:

% **$MPI_ROOT/bin/mpirun -prot -f appfile**

% **$MPI_ROOT/bin/mpirun -prot -f appfile -- 1000000**

If LSF is being used, the host names in the appfile wouldn't matter, and the command to run would be:

% **bsub pam -mpi $MPI_ROOT/bin/mpirun -prot -f appfile**

% **bsub pam -mpi $MPI_ROOT/bin/mpirun -prot -f appfile -- 1000000**

2. The `srun` command is available.

   For this case then you would run a command like this:

   % **$MPI_ROOT/bin/mpirun -prot -srun -N 8 -n 8 /path/to/pp.x**

   % **$MPI_ROOT/bin/mpirun -prot -srun -N 8 -n 8 /path/to/pp.x 1000000**

   Replacing "8" with the number of hosts.

   If LSF is being used, the command to run might be this:

   % **bsub -I -n 16 $MPI_ROOT/bin/mpirun -prot -srun /path/to/pp.x**

   % **bsub -I -n 16 $MPI_ROOT/bin/mpirun -prot -srun /path/to/pp.x 1000000**

3. The `prun` command is available.

   This case is basically identical to the `srun` case with the change of using `prun` in place of `srun`.

In each case above, the first `mpirun` uses 0-bytes of data per message and is for checking latency. The second `mpirun` uses 1000000 bytes per message and is for checking bandwidth.

```
#include <stdio.h>
#include <stdlib.h>
#ifndef _WIN32
#include <unistd.h>
#endif
#include <string.h>
#include <math.h>
#include <mpi.h>
#define NLOOPS     1000
#define ALIGN   4096
#define SEND(t) MPI_Send(buf, nbytes, MPI_CHAR, partner, (t),MPI_COMM_WORLD)
#define RECV(t) MPI_Recv(buf, nbytes, MPI_CHAR, partner, (t),MPI_COMM_WORLD, &status)
#ifdef CHECK
#  define SETBUF()for (j=0; j<nbytes; j++) {
                  buf[j] = (char) (j + i);
               }
#  define CLRBUF()memset(buf, 0, nbytes)
#  define CHKBUF()for (j = 0; j < nbytes; j++) {
                  if (buf[j] != (char) (j + i)) {
                       printf("error: buf[%d] = %d, "
                            "not %d\n",
                            j, buf[j], j + i);
                      break;
                  }
            }
#else
```

```
#  define SETBUF()
#  define CLRBUF()
#  define CHKBUF()
#endifint
main(argc, argv)int             argc;
char            *argv[];{
     int         i;
#ifdef CHECK
     int         j;#endif
     double           start, stop;
     intn        bytes = 0;
     int         rank, size;
     int         root;
     int         partner;
     MPI_Status  status;
     char        *buf, *obuf;
     char        myhost[MPI_MAX_PROCESSOR_NAME];
     int         len;
     char        str[1024];     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Get_processor_name(myhost, &len);     if (size < 2) {
        if ( ! rank) printf("rping: must have two+ processes\n");
        MPI_Finalize();
        exit(0);     }     nbytes = (argc > 1) ? atoi(argv[1]) : 0;
     if (nbytes < 0) nbytes = 0;/*
 * Page-align buffers and displace them in the cache to avoid collisions.
 */
     buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
     obuf = buf;
     if (buf == 0) {
           MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
           exit(1);
     }
     buf = (char *) ((((unsigned long) buf) + (ALIGN - 1)) & ~(ALIGN - 1));
     if (rank > 0) buf += 524288;
     memset(buf, 0, nbytes);
/*
 * Ping-pong.
 */
      for (root=0; root<size; root++) {
            if (rank == root) {
                partner = (root + 1) % size;
                sprintf(str, "[%d:%s] ping-pong %d bytes ...\n",
root, myhost, nbytes);
/*
 * warm-up loop
 */                for (i = 0; i < 5; i++) {
                      SEND(1);
                      RECV(1);
              }/*
 * timing loop
                         */
             start = MPI_Wtime();
            for (i = 0; i < NLOOPS; i++) {
                 SETBUF();
                 SEND(1000 + i);
                 CLRBUF();
                 RECV(2000 + i);
                CHKBUF();
            }
            stop = MPI_Wtime();
            sprintf(&str[strlen(str)],
                 "%d bytes: %.2f usec/msg\n", nbytes,
                 (stop - start) / NLOOPS / 2 * 1024 * 1024);
            if (nbytes > 0) {
                 sprintf(&str[strlen(str)],
                     "%d bytes: %.2f MB/sec\n", nbytes,
                     nbytes / (1024. * 1024.) /
                    ((stop - start) / NLOOPS / 2));
            }
            fflush(stdout);
      } else if (rank == (root+1)%size) {
```

```
/*
 * warm-up loop
 */
           partner = root;
           for (i = 0; i < 5; i++) {
                RECV(1);
                SEND(1);
           }
          for (i = 0; i < NLOOPS; i++) {
               CLRBUF();
               RECV(1000 + i);
               CHKBUF();
               SETBUF();
               SEND(2000 + i);
           }
      }

      MPI_Bcast(str, 1024, MPI_CHAR, root, MPI_COMM_WORLD);
      if (rank == 0) {
             printf("%s", str);
      }
   }
   free(obuf);
   MPI_Finalize();
   exit(0);}
```

ping_pong_ring.c output

Example output might look like this:

```
> Host 0 -- ip 192.168.9.10 -- ranks 0
> Host 1 -- ip 192.168.9.11 -- ranks 1
> Host 2 -- ip 192.168.9.12 -- ranks 2
> Host 3 -- ip 192.168.9.13 -- ranks 3
>
>host | 0123
> ======|====================
> 0 : SHMVAPI VAPI VAPI
> 1 : VAPI SHMVAPI VAPI
> 2 : VAPI VAPI SHMVAPI
> 3 : VAPI VAPI VAPI SHM
>
> [0:hostA] ping-pong 0 bytes ...
> 0 bytes: 4.57 usec/msg
> [1:hostB] ping-pong 0 bytes ...
> 0 bytes: 4.38 usec/msg
> [2:hostC] ping-pong 0 bytes ...
> 0 bytes: 4.42 usec/msg
> [3:hostD] ping-pong 0 bytes ...
> 0 bytes: 4.42 usec/msg
```

The table showing SHM/VAPI is printed because of the -prot option (print protocol) specified in the `mpirun` command. In general, it could show any of the following settings:

`VAPI`: InfiniBand

`UDAPL`: InfiniBand

`IBV`: InfiniBand

`PSM`: InfiniBand

`MX`: Myrinet MX

`IBAL`: InfiniBand (on Windows only)

`IT`: IT-API on InfiniBand

`GM`: Myrinet GM2

`ELAN`: Quadrics Elan4

`TCP`: TCP/IP

`MPID`: commd

`SHM`: Shared Memory (intra host only)

If the table shows TCP/IP for hosts, the host might not have the correct network drivers installed.

If a host shows considerably worse performance than another, it can often indicate a bad card or cable.

If the run aborts with an error message, Platform MPI might have determined incorrectly which interconnect was available. One common way to encounter this problem is to run a 32-bit application on a 64-bit machine like an Opteron or Intel64. It is not uncommon for network vendors for InfiniBand and others to only provide 64-bit libraries for their network.

Platform MPI makes its decision about what interconnect to use before it knows the application's bitness. To have proper network selection in that case, specify if the application is 32-bit when running on Opteron and Intel64 machines:

```
% $MPI_ROOT/bin/mpirun -mpi32 ...
```

ping_pong_ring.c (Windows)

Often, clusters might have Ethernet and some form of higher-speed interconnect such as InfiniBand. This section describes how to use the ping_pong_ring.c example program to confirm that you can run using the interconnect.

Running a test like this, especially on a new cluster, is useful to ensure that the correct network drivers are installed and that network hardware is functioning properly. If any machine has defective network cards or cables, this test can also be useful for identifying which machine has the problem.

To compile the program, set the *MPI_ROOT*environment variable to the location of Platform MPI. The default is `"C:\Program Files (x86)\Platform-MPI"` for 64-bit systems, and `"C:\Program Files\Platform-MPI"` for 32-bit systems. This might already be set by the Platform MPI installation.

Open a command window for the compiler you plan on using. This includes all libraries and compilers in path, and compile the program using the `mpicc` wrappers:

```
>"%MPI_ROOT%\bin\mpicc" -mpi64 /out:pp.exe "%MPI_ROOT%\help\ping_ping_ring.c"
```

Use the start-up for your cluster. Your situation should resemble one of the following:

1. If running on Windows HPCS using automatic scheduling:

   Submit the command to the scheduler, but include the total number of processes needed on the nodes as the -np command. This is *not* the rank count when used in this fashion. Also include the -nodexflag to indicate only one rank/node.

   Assume 4 CPUs/nodes in this cluster. The command would be:

   ```
   > "%MPI_ROOT%\bin\mpirun" -ccp -np 12 -IBAL -nodex -prot ping_ping_ring.exe
   ```

   ```
   > "%MPI_ROOT%\bin\mpirun" -ccp -np 12 -IBAL -nodex -prot ping_ping_ring.exe 10000
   ```

   In each case above, the first `mpirun` command uses 0 bytes per message and verifies latency. The second `mpirun` command uses 1000000 bytes per message and verifies bandwidth.

```
#include <stdio.h>
#include <stdlib.h>
#ifndef _WIN32
#include <unistd.h>
#endif
```

```
#include <string.h>
#include <math.h>
#include <mpi.h>#define NLOOPS     1000
#define ALIGN      4096
#define SEND(t) MPI_Send(buf, nbytes, MPI_CHAR, partner, (t), MPI_COMM_WORLD)
#define RECV(t) MPI_Recv(buf, nbytes, MPI_CHAR, partner, (t), MPI_COMM_WORLD,
&status)
#ifdef CHECK
#  define SETBUF()for (j=0; j<nbytes; j++) {
                  buf[j] = (char) (j + i);
             }#  define CLRBUF()memset(buf, 0, nbytes)
#  define CHKBUF()for (j = 0; j < nbytes; j++) {
                  if (buf[j] != (char) (j + i)) {
                       printf("error: buf[%d] = %d, "
                            "not %d\n",
                            j, buf[j], j + i);
                       break;
                  }
             }
#else
#  define SETBUF()
#  define CLRBUF()
#  define CHKBUF()
#endifint
main(argc, argv)int              argc;
char            *argv[];{
     int         i;
#ifdef CHECK
     int         j;#endif
     double           start, stop;
     intn        bytes = 0;
     int         rank, size;
     int         root;
     int         partner;
     MPI_Status  status;
     char        *buf, *obuf;
     char        myhost[MPI_MAX_PROCESSOR_NAME];
     int         len;
     char        str[1024];     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Get_processor_name(myhost, &len);     if (size < 2) {
        if ( ! rank) printf("rping: must have two+ processes\n");
        MPI_Finalize();
        exit(0);     }     nbytes = (argc > 1) ? atoi(argv[1]) : 0;
     if (nbytes < 0) nbytes = 0;/*
 * Page-align buffers and displace them in the cache to avoid  collisions.
 */
     buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
     obuf = buf;
     if (buf == 0) {
           MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
                                                exit(1);
     }
                                          buf = (char *) ((((unsigned long) buf) +
(ALIGN - 1)) & ~(ALIGN - 1));
                                          if (rank > 0) buf += 524288;
     memset(buf, 0, nbytes);
/*
 * Ping-pong.
 */
      for (root=0; root<size; root++) {
                                                 if (rank == root) {
                partner = (root + 1) % size;
                sprintf(str, "[%d:%s] ping-pong %d bytes ...\n",
root, myhost, nbytes);
/*
 * warm-up loop
 */                for (i = 0; i < 5; i++) {
                      SEND(1);
                      RECV(1);
             }/*
 * timing loop
```

```
 */
            start = MPI_Wtime();
           for (i = 0; i < NLOOPS; i++) {
                SETBUF();
                SEND(1000 + i);
                CLRBUF();
                RECV(2000 + i);
               CHKBUF();
           }
           stop = MPI_Wtime();
           sprintf(&str[strlen(str)],
                "%d bytes: %.2f usec/msg\n", nbytes,
                (stop - start) / NLOOPS / 2 * 1024 * 1024);
           if (nbytes > 0) {
                sprintf(&str[strlen(str)],
                    "%d bytes: %.2f MB/sec\n", nbytes,
                    nbytes / (1024. * 1024.) /
                   ((stop - start) / NLOOPS / 2));
           }
           fflush(stdout);
      } else if (rank == (root+1)%size) {
/*
 * warm-up loop
 */
           partner = root;
           for (i = 0; i < 5; i++) {
                RECV(1);
                SEND(1);
           }
          for (i = 0; i < NLOOPS; i++) {
                CLRBUF();
                RECV(1000 + i);
                CHKBUF();
                SETBUF();
                SEND(2000 + i);
           }
      }

      MPI_Bcast(str, 1024, MPI_CHAR, root, MPI_COMM_WORLD);
      if (rank == 0) {
             printf("%s", str);
      }
   }
   free(obuf);
   MPI_Finalize();
   exit(0);}
```

ping_pong_ring.c output

Example output might look like this:

```
Host 0 -- ip 172.16.159.3 -- ranks 0
Host 1 -- ip 172.16.150.23 -- ranks 1
Host 2 -- ip 172.16.150.24 -- ranks 2
host | 0    1    2
                                  =====|================
   0 : SHM  IBAL IBAL
   1 : IBAL SHM  IBAL
   2 : IBAL IBAL SHM
[0:mpiccp3] ping-pong 1000000 bytes ...
1000000 bytes: 1089.29 usec/msg
1000000 bytes: 918.03 MB/sec
[1:mpiccp4] ping-pong 1000000 bytes ...
1000000 bytes: 1091.99 usec/msg
1000000 bytes: 915.76 MB/sec
[2:mpiccp5] ping-pong 1000000 bytes ...
1000000 bytes: 1084.63 usec/msg
1000000 bytes: 921.97 MB/sec/
```

The table showing SHM/IBAL is printed because of the -prot option (print protocol) specified in the `mpirun` command.

It could show any of the following settings:

- `IBAL`: IBAL on InfiniBand
- `MX`: Myrinet Express
- `TCP`: TCP/IP
- `MPID`: daemon communication mode
- `SHM`: shared memory (intra host only)

If a host shows considerably worse performance than another, it can often indicate a bad card or cable.

If the run aborts with an error message, Platform MPI might have incorrectly determined which interconnect was available.

compute_pi.f

This Fortran 77 example computes pi by integrating f(x) = 4/(1 + x*x). Each process:

1. Receives the number of intervals used in the approximation
2. Calculates the areas of its rectangles
3. Synchronizes for a global summation

Process 0 prints the result of the calculation.

```
program main
include 'mpif.h'
double precision PI25DT
parameter(PI25DT = 3.14159265358979323846264338327950288419716939937510d0)
double precision  mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
C
C Function to integrate
C
f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
print *, "Process ", myid, " of ", numprocs, " is alive"
sizetype = 1
sumtype = 2
if (myid .eq. 0) then
n = 100
endif
call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
C
C Calculate the interval size.
C
h = 1.0d0 / n
sum  = 0.0d0
do 20 i = myid + 1, n, numprocs
x = h * (dble(i) - 0.5d0)
sum = sum + f(x)
20   continue
mypi = h * sum
C
C Collect all the partial sums.
C
call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
+               MPI_SUM, 0, MPI_COMM_WORLD, ierr)
C
C Process 0 prints the result.
C
if (myid .eq. 0) then
write(6, 97) pi, abs(pi - PI25DT)
97           format('  pi is approximately: ', F18.16,
+              '  Error is: ', F18.16)
endif
```

```
call MPI_FINALIZE(ierr)
stop
end
```

compute_pi output

The output from running the compute_pi executable is shown below. The application was run with -np=10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  2 of  10 is alive
Process  3 of  10 is alive
Process  4 of  10 is alive
Process  5 of  10 is alive
Process  6 of  10 is alive
Process  7 of  10 is alive
Process  8 of  10 is alive
Process  9 of  10 is alive
pi is approximately: 3.1416009869231249
Error is: 0.0000083333333318
```

master_worker.f90

In this Fortran 90 example, a master task initiates (numtasks - 1) number of worker tasks. The master distributes an equal portion of an array to each worker task. Each worker task receives its portion of the array and sets the value of each element to (the element's index + 1). Each worker task then sends its portion of the modified array back to the master.

```
program array_manipulation
include 'mpif.h'
integer (kind=4) :: status(MPI_STATUS_SIZE)
integer (kind=4), parameter :: ARRAYSIZE = 10000, MASTER = 0
integer (kind=4) :: numtasks, numworkers, taskid, dest, index, i
integer (kind=4) :: arraymsg, indexmsg, source, chunksize, int4, real4
real (kind=4) :: data(ARRAYSIZE), result(ARRAYSIZE)
integer (kind=4) :: numfail, ierr
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, taskid, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, numtasks, ierr)
numworkers = numtasks - 1
chunksize = (ARRAYSIZE / numworkers)
arraymsg = 1
indexmsg = 2
int4 = 4
real4 = 4
numfail = 0
! ****************************** Master task ****************************
if (taskid .eq. MASTER) then
data = 0.0
index = 1
do dest = 1, numworkers
call MPI_Send(index, 1, MPI_INTEGER, dest, 0, MPI_COMM_WORLD, ierr)
call MPI_Send(data(index), chunksize, MPI_REAL, dest, 0, &
MPI_COMM_WORLD, ierr)
index = index + chunksize
end do
do i = 1, numworkers
source = i
call MPI_Recv(index, 1, MPI_INTEGER, source, 1, MPI_COMM_WORLD, &
status, ierr)
call MPI_Recv(result(index), chunksize, MPI_REAL, source, 1, &
MPI_COMM_WORLD, status, ierr)
end do
do i = 1, numworkers*chunksize
if (result(i) .ne. (i+1)) then
```

```
codeph>print *, 'element ', i, ' expecting ', (i+1), ' actual is ', result
(i)
numfail = numfail + 1
endif
enddo
if (numfail .ne. 0) then
print *, 'out of ', ARRAYSIZE, ' elements, ', numfail, ' wrong answers'
else
print *, 'correct results!'
endif
end if
! ****************************** Worker task ******************************
if (taskid .gt. MASTER) then
call MPI_Recv(index, 1, MPI_INTEGER, MASTER, 0, MPI_COMM_WORLD, &
status, ierr)
call MPI_Recv(result(index), chunksize, MPI_REAL, MASTER, 0, &
MPI_COMM_WORLD, status, ierr)
do i = index, index + chunksize - 1
result(i) = i + 1
end do
call MPI_Send(index, 1, MPI_INTEGER, MASTER, 1, MPI_COMM_WORLD, ierr)
call MPI_Send(result(index), chunksize, MPI_REAL, MASTER, 1, &
MPI_COMM_WORLD, ierr)
end if
call MPI_Finalize(ierr)
end program array_manipulation
```

master_worker output

The output from running the master_worker executable is shown below. The application was run with -np=2.

```
correct results!
```

cart.C

This C++ program generates a virtual topology. The class Node represents a node in a 2-D torus. Each process is assigned a node or nothing. Each node holds integer data, and the shift operation exchanges the data with its neighbors. Thus, north-east-south-west shifting returns the initial data.

```
#include <stdio.h>
#include <mpi.h>
#define NDIMS   2
typedef enum { NORTH, SOUTH, EAST, WEST } Direction;
// A node in 2-D torus
class Node {
private:
MPI_Comm      comm;
int           dims[NDIMS], coords[NDIMS];
int           grank, lrank;
int           data;
public:
Node(void);
~Node(void);
void profile(void);
void print(void);
void shift(Direction);
};
// A constructor
Node::Node(void)
{
int i, nnodes, periods[NDIMS];
// Create a balanced distribution
MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
for (i = 0; i < NDIMS; i++) { dims[i] = 0; }
MPI_Dims_create(nnodes, NDIMS, dims);
// Establish a cartesian topology communicator
```

```
for (i = 0; i < NDIMS; i++) { periods[i] = 1; }
MPI_Cart_create(MPI_COMM_WORLD, NDIMS, dims, periods, 1, &comm);
// Initialize the data
MPI_Comm_rank(MPI_COMM_WORLD, &grank);
if (comm == MPI_COMM_NULL) {
lrank = MPI_PROC_NULL;
data = -1;
} else {
MPI_Comm_rank(comm, &lrank);
data = lrank;
MPI_Cart_coords(comm, lrank, NDIMS, coords);
}
}
// A destructor
Node::~Node(void)
{
if (comm != MPI_COMM_NULL) {
MPI_Comm_free(&comm);
}
}
// Shift function
void Node::shift(Direction dir)
{
if (comm == MPI_COMM_NULL) { return; }
int direction, disp, src, dest;
if (dir == NORTH) {
direction = 0; disp = -1;
} else if (dir == SOUTH) {
direction = 0; disp = 1;
} else if (dir == EAST) {
direction = 1; disp = 1;
} else {
direction = 1; disp = -1;
}
MPI_Cart_shift(comm, direction, disp, &src, &dest);
MPI_Status stat;
MPI_Sendrecv_replace(&data, 1, MPI_INT, dest, 0, src, 0, comm, &stat);
}
// Synchronize and print the data being held
void Node::print(void)
{
if (comm != MPI_COMM_NULL) {
MPI_Barrier(comm);
if (lrank == 0) { puts(""); } // line feed
MPI_Barrier(comm);
printf("(%d, %d) holds %d\n", coords[0], coords[1], data);
}
}
// Print object's profile
void Node::profile(void)
{
// Non-member does nothing
if (comm == MPI_COMM_NULL) { return; }
// Print "Dimensions" at first
if (lrank == 0) {
printf("Dimensions: (%d, %d)\n", dims[0], dims[1]);
}
MPI_Barrier(comm);
// Each process prints its profile
printf("global rank %d: cartesian rank %d, coordinate (%d, %d)\n",
grank, lrank, coords[0], coords[1]);
}
// Program body
//
// Define a torus topology and demonstrate shift operations.
//
                        void body(void)
{
Node node;
node.profile();
                        node.print();
node.shift(NORTH);
node.print();
```

```
node.shift(EAST);
                        node.print();
node.shift(SOUTH);
node.print();
node.shift(WEST);
node.print();
}
//
// Main program---it is probably a good programming practice to call
//   MPI_Init() and MPI_Finalize() here.
//
int main(int argc, char **argv)
{
MPI_Init(&argc, &argv);
body();
MPI_Finalize();
}
```

cart output

The output from running the cart executable is shown below. The application was run with `-np=4`.

```
Dimensions: (2, 2)
global rank 0: cartesian rank 0, coordinate (0, 0)
global rank 1: cartesian rank 1, coordinate (0, 1)
global rank 3: cartesian rank 3, coordinate (1, 1)
global rank 2: cartesian rank 2, coordinate (1, 0)
(0, 0) holds 0
(1, 0) holds 2
(1, 1) holds 3
(0, 1) holds 1
(0, 0) holds 2
(1, 0) holds 0
(0, 1) holds 3
(1, 1) holds 1
(0, 0) holds 3
(0, 1) holds 2
(1, 0) holds 1
(1, 1) holds 0
(0, 0) holds 1
(1, 0) holds 3
(0, 1) holds 0
(1, 1) holds 2
(0, 0) holds 0
(1, 0) holds 2
(0, 1) holds 1
(1, 1) holds 3
```

communicator.c

This C example shows how to make a copy of the default communicator *MPI_COMM_WORLD* using MPI_Comm_dup.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int
main(argc, argv)
int                     argc;
char                    *argv[];
{
int             rank, size, data;
MPI_Status      status;
MPI_Comm        libcomm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size != 2) {
if ( ! rank) printf("communicator: must have two processes\n");
```

```
MPI_Finalize();
exit(0);
}
MPI_Comm_dup(MPI_COMM_WORLD, &libcomm);
if (rank == 0) {
data = 12345;
MPI_Send(&data, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
data = 6789;
MPI_Send(&data, 1, MPI_INT, 1, 5, libcomm);
} else {
MPI_Recv(&data, 1, MPI_INT, 0, 5, libcomm, &status);
printf("received libcomm data = %d\n", data);
MPI_Recv(&data, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status);
printf("received data = %d\n", data);
}
MPI_Comm_free(&libcomm);
MPI_Finalize();
return(0);
}
```

communicator output

The output from running the communicator executable is shown below. The application was run with -np=2.

```
received libcomm data = 6789
received data = 12345
```

multi_par.f

The Alternating Direction Iterative (ADI) method is often used to solve differential equations. In this example, multi_par.f, a compiler that supports OPENMP directives is required in order to achieve multi-level parallelism. multi_par.f implements the following logic for a 2-dimensional compute region:

```
      DO J=1,JMAX
DO I=2,IMAX
A(I,J)=A(I,J)+A(I-1,J)
ENDDO
ENDDO
                              DO J=2,JMAX
DO I=1,IMAX
A(I,J)=A(I,J)+A(I,J-1)
ENDDO
ENDDO
```

There are loop carried dependencies on the first dimension (array's row) in the first innermost DO loop and the second dimension (array's column) in the second outermost DO loop.

A simple method for parallelizing the fist outer-loop implies a partitioning of the array in column blocks, while another for the second outer-loop implies a partitioning of the array in row blocks.

With message-passing programming, such a method requires massive data exchange among processes because of the partitioning change. "Twisted data layout" partitioning is better in this case because the partitioning used for the parallelization of the first outer-loop can accommodate the other of the second outer-loop.

Select Network Service
Click the Network Service that you want to install, then click OK.
Network Service:
Myricom Myrinet MX Mapper Daemon
This driver is not digitally signed!
Tell me why driver signing is important
OK
Cancel

Figure 2: Array partitioning

In this sample program, the rank *n*process is assigned to the partition *n* at distribution initialization. Because these partitions are not contiguous-memory regions, MPI's derived datatype is used to define the partition layout to the MPI system.

Each process starts with computing summations in row-wise fashion. For example, the rank 2 process starts with the block that is on the 0th-row block and 2nd-column block (denoted as [0,2]).

The block computed in the second step is [1,3]. Computing the first row elements in this block requires the last row elements in the [0,3] block (computed in the first step in the rank 3 process). Thus, the rank 2 process receives the data from the rank 3 process at the beginning of the second step. The rank 2 process also sends the last row elements of the [0,2] block to the rank 1 process that computes [1,2] in the second step. By repeating these steps, all processes finish summations in row-wise fashion (the first outer-loop in the illustrated program).

The second outer-loop (the summations in column-wise fashion) is done in the same manner. For example, at the beginning of the second step for the column-wise summations, the rank 2 process receives data from the rank 1 process that computed the [3,0] block. The rank 2 process also sends the last column of the [2,0] block to the rank 3 process. Each process keeps the same blocks for both of the outer-loop computations.

This approach is good for distributed memory architectures where repartitioning requires massive data communications that are expensive. However, on shared memory architectures, the partitioning of the compute region does not imply data distribution. The row- and column-block partitioning method requires just one synchronization at the end of each outer loop.

For distributed shared-memory architectures, the mix of the two methods can be effective. The sample program implements the twisted-data layout method with MPI and the row- and column-block

partitioning method with OPENMP thread directives. In the first case, the data dependency is easily satisfied because each thread computes down a different set of columns. In the second case we still want to compute down the columns for cache reasons, but to satisfy the data dependency, each thread computes a different portion of the same column and the threads work left to right across the rows together.

```
      implicit none
      include 'mpif.h'
      integer nrow                      ! # of rows
      integer ncol                      ! # of columns
      parameter(nrow=1000,ncol=1000)
      double precision array(nrow,ncol) ! compute region
      integer blk                       ! block iteration counter
      integer rb                        ! row block number
      integer cb                        ! column block number
      integer nrb                       ! next row block number
      integer ncb                       ! next column block number
      integer rbs(:)                    ! row block start subscripts
      integer rbe(:)                    ! row block end subscripts
      integer cbs(:)                    ! column block start subscripts
      integer cbe(:)                    ! column block end subscripts
      integer rdtype(:)                 ! row block communication datatypes
      integer cdtype(:)                 ! column block communication datatypes
      integer twdtype(:)                ! twisted distribution datatypes
      integer ablen(:)                  ! array of block lengths
      integer adisp(:)                  ! array of displacements
      integer adtype(:)                 ! array of datatypes
      allocatable rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype,ablen,adisp,adtype
      integer rank                      ! rank iteration counter
      integer comm_size                 ! number of MPI processes
      integer comm_rank                 ! sequential ID of MPI process
      integer ierr                      ! MPI error code
      integer mstat(mpi_status_size)    ! MPI function status
      integer src                       ! source rank
      integer dest                      ! destination rank
      integer dsize                     ! size of double precision in bytes
   double precision startt,endt,elapsed ! time keepers
      external compcolumn,comprow       ! subroutines execute in threads

c
c     MPI initialization
c
      call mpi_init(ierr)
      call mpi_comm_size(mpi_comm_world,comm_size,ierr)
      call mpi_comm_rank(mpi_comm_world,comm_rank,ierr)

c
c     Data initialization and start up
c
      if (comm_rank.eq.0) then
         write(6,*) 'Initializing',nrow,' x',ncol,' array...'
         call getdata(nrow,ncol,array)
         write(6,*) 'Start computation'
      endif
      call mpi_barrier(MPI_COMM_WORLD,ierr)
      startt=mpi_wtime()
c
c     Compose MPI datatypes for row/column send-receive
c
c     Note that the numbers from rbs(i) to rbe(i) are the indices
c     of the rows belonging to the i'th block of rows.  These indices
c     specify a portion (the i'th portion) of a column and the
c     datatype rdtype(i) is created as an MPI contiguous datatype
c     to refer to the i'th portion of a column.  Note this is a
c     contiguous datatype because fortran arrays are stored
c     column-wise.
c
c     For a range of columns to specify portions of rows, the situation
c     is similar:  the numbers from cbs(j) to cbe(j) are the indices
c     of the columns belonging to the j'th block of columns.  These
c     indices specify a portion (the j'th portion) of a row, and the
c     datatype cdtype(j) is created as an MPI vector datatype to refer
c     to the j'th portion of a row.  Note this a vector datatype
```

```
c     because adjacent elements in a row are actually spaced nrow
c     elements apart in memory.
c
      allocate(rbs(0:comm_size-1),rbe(0:comm_size-1),cbs(0:comm_size-1),
     *     cbe(0:comm_size-1),rdtype(0:comm_size-1),
     *     cdtype(0:comm_size-1),twdtype(0:comm_size-1))
      do blk=0,comm_size-1
         call blockasgn(1,nrow,comm_size,blk,rbs(blk),rbe(blk))
         call mpi_type_contiguous(rbe(blk)-rbs(blk)+1,
     *        mpi_double_precision,rdtype(blk),ierr)
         call mpi_type_commit(rdtype(blk),ierr)
         call blockasgn(1,ncol,comm_size,blk,cbs(blk),cbe(blk))
         call mpi_type_vector(cbe(blk)-cbs(blk)+1,1,nrow,
     *        mpi_double_precision,cdtype(blk),ierr)
         call mpi_type_commit(cdtype(blk),ierr)
      enddo



c     Compose MPI datatypes for gather/scatter
c
c     Each block of the partitioning is defined as a set of fixed length
c     vectors.  Each process'es partition is defined as a struct of such
c     blocks.
c
      allocate(adtype(0:comm_size-1),adisp(0:comm_size-1),
     *     ablen(0:comm_size-1))
      call mpi_type_extent(mpi_double_precision,dsize,ierr)
      do rank=0,comm_size-1
         do rb=0,comm_size-1
            cb=mod(rb+rank,comm_size)
            call mpi_type_vector(cbe(cb)-cbs(cb)+1,rbe(rb)-rbs(rb)+1,
     *           nrow,mpi_double_precision,adtype(rb),ierr)
            call mpi_type_commit(adtype(rb),ierr)
            adisp(rb)=((rbs(rb)-1)+(cbs(cb)-1)*nrow)*dsize
            ablen(rb)=1
         enddo
         call mpi_type_struct(comm_size,ablen,adisp,adtype,
     *        twdtype(rank),ierr)
         call mpi_type_commit(twdtype(rank),ierr)
         do rb=0,comm_size-1
            call mpi_type_free(adtype(rb),ierr)
         enddo
      enddo
      deallocate(adtype,adisp,ablen)

c     Scatter initial data with using derived datatypes defined above
c     for the partitioning.  MPI_send() and MPI_recv() will find out the
c     layout of the data from those datatypes.  This saves application
c     programs to manually pack/unpack the data, and more importantly,
c     gives opportunities to the MPI system for optimal communication
c     strategies.
c
      if (comm_rank.eq.0) then
         do dest=1,comm_size-1
            call mpi_send(array,1,twdtype(dest),dest,0,mpi_comm_world,
     *           ierr)
         enddo
      else
         call mpi_recv(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        mstat,ierr)
      endif

c
c     Computation
c
c     Sum up in each column.
c     Each MPI process, or a rank, computes blocks that it is assigned.
c     The column block number is assigned in the variable 'cb'.  The
c     starting and ending subscripts of the column block 'cb' are
c     stored in 'cbs(cb)' and 'cbe(cb)', respectively.  The row block
c     number is assigned in the variable 'rb'.  The starting and ending
c     subscripts of the row block 'rb' are stored in 'rbs(rb)' and
```

```
c     'rbe(rb)', respectively, as well.
      src=mod(comm_rank+1,comm_size)
      dest=mod(comm_rank-1+comm_size,comm_size)
      ncb=comm_rank
      do rb=0,comm_size-1
         cb=ncb
c
c     Compute a block.  The function will go thread-parallel if the
c     compiler supports OPENMP directives.
c
         call compcolumn(nrow,ncol,array,
     *                   rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (rb.lt.comm_size-1) then
c
c     Send the last row of the block to the rank that is to compute the
c     block next to the computed block.  Receive the last row of the
c     block that the next block being computed depends on.
c
            nrb=rb+1
            ncb=mod(nrb+comm_rank,comm_size)
            call mpi_sendrecv(array(rbe(rb),cbs(cb)),1,cdtype(cb),dest,
     *           0,array(rbs(nrb)-1,cbs(ncb)),1,cdtype(ncb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo
c
c     Sum up in each row.
c     The same logic as the loop above except rows and columns are
c     switched.
c
      src=mod(comm_rank-1+comm_size,comm_size)
      dest=mod(comm_rank+1,comm_size)
      do cb=0,comm_size-1
         rb=mod(cb-comm_rank+comm_size,comm_size)
         call comprow(nrow,ncol,array,
     *                rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (cb.lt.comm_size-1) then
            ncb=cb+1
            nrb=mod(ncb-comm_rank+comm_size,comm_size)
            call mpi_sendrecv(array(rbs(rb),cbe(cb)),1,rdtype(rb),dest,
     *           0,array(rbs(nrb),cbs(ncb)-1),1,rdtype(nrb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo
c
c     Gather computation results
c
      call mpi_barrier(MPI_COMM_WORLD,ierr)
      endt=mpi_wtime()

      if (comm_rank.eq.0) then
         do src=1,comm_size-1
            call mpi_recv(array,1,twdtype(src),src,0,mpi_comm_world,
     *           mstat,ierr)
         enddo

elapsed=endt-startt
         write(6,*) 'Computation took',elapsed,' seconds'
      else
         call mpi_send(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        ierr)
      endif
c
c     Dump to a file
c
c      if (comm_rank.eq.0) then
c         print*,'Dumping to adi.out...'
c         open(8,file='adi.out')
c         write(8,*) array
c         close(8,status='keep')
c      endif
c
c     Free the resources
```

```
c
      do rank=0,comm_size-1
         call mpi_type_free(twdtype(rank),ierr)
      enddo
      do blk=0,comm_size-1
         call mpi_type_free(rdtype(blk),ierr)
         call mpi_type_free(cdtype(blk),ierr)
      enddo
      deallocate(rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype)
c
c     Finalize the MPI system
c
      call mpi_finalize(ierr)
      end
c**********************************************************************
      subroutine blockasgn(subs,sube,blockcnt,nth,blocks,blocke)
c
c     This subroutine:
c     is given a range of subscript and the total number of blocks in
c     which the range is to be divided, assigns a subrange to the caller
c     that is n-th member of the blocks.
c
      implicit none
      integer subs           ! (in)      subscript start
      integer sube           ! (in)      subscript end
      integer blockcnt       ! (in)      block count
      integer nth            ! (in)      my block (begin from 0)
      integer blocks         ! (out)     assigned block start subscript
      integer blocke         ! (out)     assigned block end subscript
c
      integer d1,m1
c
      d1=(sube-subs+1)/blockcnt
      m1=mod(sube-subs+1,blockcnt)
      blocks=nth*d1+subs+min(nth,m1)
      blocke=blocks+d1-1
      if(m1.gt.nth)blocke=blocke+1
      end
c
c**********************************************************************
      subroutine compcolumn(nrow,ncol,array,rbs,rbe,cbs,cbe)
c
c     This subroutine:
c     does summations of columns in a thread.
c
      implicit none

      integer nrow                      ! # of rows
      integer ncol                      ! # of columns
      double precision array(nrow,ncol)  ! compute region
      integer rbs                       ! row block start subscript
      integer rbe                       ! row block end subscript
      integer cbs                       ! column block start subscript
      integer cbe                       ! column block end subscript

c
c     Local variables
c
      integer i,j

c
c     The OPENMP directive below allows the compiler to split the
c     values for "j" between a number of threads.  By making i and j
c     private, each thread works on its own range of columns "j",
c     and works down each column at its own pace "i".
c
c     Note no data dependency problems arise by having the threads all
c     working on different columns simultaneously.
c

C$OMP PARALLEL DO PRIVATE(i,j)
      do j=cbs,cbe
         do i=max(2,rbs),rbe
```

```
            array(i,j)=array(i-1,j)+array(i,j)
         enddo
      enddo
C$OMP END PARALLEL DO
      end

c**********************************************************************
      subroutine comprow(nrow,ncol,array,rbs,rbe,cbs,cbe)
c
c     This subroutine:
c     does summations of rows in a thread.
c
      implicit none

      integer nrow                       ! # of rows
      integer ncol                       ! # of columns
      double precision array(nrow,ncol)  ! compute region
      integer rbs                        ! row block start subscript
      integer rbe                        ! row block end subscript
      integer cbs                        ! column block start subscript
      integer cbe                        ! column block end subscript

c
c     Local variables
c
      integer i,j

c
c     The OPENMP directives below allow the compiler to split the
c     values for "i" between a number of threads, while "j" moves
c     forward lock-step between the threads.  By making j shared
c     and i private, all the threads work on the same column "j" at
c     any given time, but they each work on a different portion "i"
c     of that column.
c
c     This is not as efficient as found in the compcolumn subroutine,
c     but is necessary due to data dependencies.
c


C$OMP PARALLEL PRIVATE(i)
      do j=max(2,cbs),cbe
C$OMP DO
         do i=rbs,rbe
            array(i,j)=array(i,j-1)+array(i,j)
         enddo
C$OMP END DO
      enddo
C$OMP END PARALLEL

      end
c
c**********************************************************************
       subroutine getdata(nrow,ncol,array)
c
c     Enter dummy data
c
      integer nrow,ncol
      double precision array(nrow,ncol)
c
      do j=1,ncol
         do i=1,nrow
            array(i,j)=(j-1.0)*ncol+i
         enddo
      enddo
      end
```

multi_par.f output

The output from running the multi_par.f executable is shown below. The application was run with -np1.

```
Initializing 1000  x 1000  array...
Start computation
Computation took 4.088211059570312E-02  seconds
```

io.c

In this C example, each process writes to a separate file called iodata*x*, where*x* represents each process rank in turn. Then, the data in iodata*x* is read back.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>

#define SIZE (65536)
#define FILENAME "iodata"

/*Each process writes to separate files and reads them back. The file name is
 iodata and the process rank is appended to it.*/

main(argc, argv)

     int argc;
     char **argv;

{
        int *buf, i, rank, nints, len, flag;
        char *filename;
        MPI_File fh;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        buf = (int *) malloc(SIZE);
        nints = SIZE/sizeof(int);
        for (i=0; i<nints; i++) buf[i] = rank*100000 + i;

        /* each process opens a separate file called FILENAME.'myrank' */

        filename = (char *) malloc(strlen(FILENAME) + 10);
        sprintf(filename, "%s.%d", FILENAME, rank);

        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);

        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_write(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* reopen the file and read the data back */

        for (i=0; i<nints; i++) buf[i] = 0;
        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);
        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_read(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* check if the data read is correct */
```

```
        flag = 0;
        for (i=0; i<nints; i++)
                if (buf[i] != (rank*100000 + i)) {
                        printf("Process %d: error, read %d, should be %d\n",
                               rank, buf[i], rank*100000+i);
                        flag = 1;
                }

        if (!flag) {
                printf("Process %d: data read back is correct\n", rank);
                MPI_File_delete(filename, MPI_INFO_NULL);
        }

        free(buf);
        free(filename);

        MPI_Finalize();
        exit(0);
}
```

io Output

The output from running the io executable is shown below. The applicat,ion was run with -np=4.

```
Process 0: data read back is correct
Process 1: data read back is correct
Process 2: data read back is correct
Process 3: data read back is correct
```

thread_safe.c

In this C example, N clients loop *MAX_WORK* times. As part of a single work item, a client must request service from one of Nservers at random. Each server keeps a count of the requests handled and prints a log of the requests to stdout. After all clients finish, the servers are shut down.

```
#include <stdio.h>
#include <mpi.h>
#include <pthread.h>

#define MAX_WORK        40
#define SERVER_TAG      88
#define CLIENT_TAG      99
#define REQ_SHUTDOWN    -1

static int service_cnt = 0;

int process_request(request)
int request;
{
      if (request != REQ_SHUTDOWN) service_cnt++;
      return request;
}

void* server(args)
void *args;

{
        int rank, request;
        MPI_Status status;
        rank = *((int*)args);

        while (1) {
                MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
                         SERVER_TAG, MPI_COMM_WORLD, &status);

                if (process_request(request) == REQ_SHUTDOWN)
                      break;

                    MPI_Send(&rank, 1, MPI_INT, status.MPI_SOURCE,
```

```
                            CLIENT_TAG, MPI_COMM_WORLD);

                printf("server [%d]: processed request %d for client %d\n",
                       rank, request, status.MPI_SOURCE);
        }

        printf("server [%d]: total service requests: %d\n", rank, service_cnt);
        return (void*) 0;
}

void client(rank, size)
int rank;
int size;

{
        int w, server, ack;
        MPI_Status status;

        for (w = 0; w < MAX_WORK; w++) {
               server = rand()%size;

        MPI_Sendrecv(&rank, 1, MPI_INT, server, SERVER_TAG, &ack,
                     1,MPI_INT,server,CLIENT_TAG,MPI_COMM_WORLD, &status);

        if (ack != server) {
                   printf("server failed to process my request\n");
                   MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
                }
        }
}

void shutdown_servers(rank)
int rank;

{
         int request_shutdown = REQ_SHUTDOWN;
         MPI_Barrier(MPI_COMM_WORLD);
         MPI_Send(&request_shutdown, 1, MPI_INT, rank,
             SERVER_TAG, MPI_COMM_WORLD);
}

main(argc, argv)
int argc;
char *argv[];

{
        int rank, size, rtn;
        pthread_t mtid;
        MPI_Status      status;
        int my_value, his_value;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        rtn = pthread_create(&mtid, 0, server, (void*)&rank);
        if (rtn != 0) {
                printf("pthread_create failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

        client(rank, size);
        shutdown_servers(rank);

        rtn = pthread_join(mtid, 0);
        if (rtn != 0) {
                printf("pthread_join failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

        MPI_Finalize();
        exit(0);
        }
```

thread_safe output

The output from running the thread_safe executable is shown below. The application was run with -np=2.

```
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
```

```
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: total service requests: 38
server [1]: total service requests: 42
```

sort.C

This program does a simple integer sort in parallel. The sort input is built using the "rand" random number generator. The program is self-checking and can run with any number of ranks.

```
#define NUM_OF_ENTRIES_PER_RANK100

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <mpi.h>

#include <limits.h>

#include <iostream.h>
#include <fstream.h>


//
// Class declarations.
//



class Entry {
private:
 int value;

public:

  Entry()
         { value = 0; }

  Entry(int x)
         { value = x; }

  Entry(const Entry &e)
         { value = e.getValue(); }

  Entry& operator= (const Entry &e)
         { value = e.getValue(); return (*this); }

  int getValue() const { return value; }

  int operator> (const Entry &e) const
         { return (value > e.getValue()); }
};


class BlockOfEntries {
private:
  Entry **entries;
  int numOfEntries;

public:
```

```
  BlockOfEntries(int *numOfEntries_p, int offset);
  ~BlockOfEntries();
  int getnumOfEntries()
       { return numOfEntries; }
  void setLeftShadow(const Entry &e)
       { *(entries[0]) = e; }
  void setRightShadow(const Entry &e)
       { *(entries[numOfEntries-1]) = e; }

  const Entry& getLeftEnd()
       { return *(entries[1]); }
  const Entry& getRightEnd()
       { return *(entries[numOfEntries-2]); }

  void singleStepOddEntries();
  void singleStepEvenEntries();
  void verifyEntries(int myRank, int baseLine);
  void printEntries(int myRank);
};



//
// Class member definitions.
//
const Entry MAXENTRY(INT_MAX);
const Entry MINENTRY(INT_MIN);



//
//BlockOfEntries::BlockOfEntries
//
//Function:- create the block of entries.
//
BlockOfEntries::BlockOfEntries(int *numOfEntries_p, int myRank)


{
//
// Initialize the random number generator's seed based on the caller's rank;
// thus, each rank should (but might not) get different random values.
//



  srand((unsigned int) myRank);

  numOfEntries = NUM_OF_ENTRIES_PER_RANK;
  *numOfEntries_p = numOfEntries;



//
// Add in the left and right shadow entries.
//

  numOfEntries += 2;



//
// Allocate space for the entries and use rand to initialize the values.
//



  entries = new Entry *[numOfEntries];
  for(int i = 1; i < numOfEntries-1; i++) {
        entries[i] = new Entry;
       *(entries[i]) = (rand()%1000) * ((rand()%2 == 0)? 1 : -1);
}
```

```
//
// Initialize the shadow entries.
//
    entries[0] = new Entry(MINENTRY);
    entries[numOfEntries-1] = new Entry(MAXENTRY);
}



//
//BlockOfEntries::~BlockOfEntries
//
//Function:- delete the block of entries.
//
BlockOfEntries::~BlockOfEntries()

{
  for(int i = 1; i < numOfEntries-1; i++) {
          delete entries[i];
  }
  delete entries[0];
  delete entries[numOfEntries-1];
  delete [] entries;
}



//
//BlockOfEntries::singleStepOddEntries
//
//Function: - Adjust the odd entries.
//
void
BlockOfEntries::singleStepOddEntries()


{
for(int i = 0; i < numOfEntries-1; i += 2) {
       if (*(entries[i]) > *(entries[i+1]) ) {
       Entry *temp = entries[i+1];
       entries[i+1] = entries[i];
       entries[i] = temp;
    }
  }
}



//
//BlockOfEntries::singleStepEvenEntries
//
//Function: - Adjust the even entries.
//
void
BlockOfEntries::singleStepEvenEntries()


{

   for(int i = 1; i < numOfEntries-2; i += 2) {
        if (*(entries[i]) > *(entries[i+1]) ) {
        Entry *temp = entries[i+1];
        entries[i+1] = entries[i];
        entries[i] = temp;
    }
  }
}



//
```

```
//BlockOfEntries::verifyEntries
//
//Function: - Verify that the block of entries for rank myRank
//  is sorted and each entry value is greater than
//  or equal to argument baseLine.
//


void
BlockOfEntries::verifyEntries(int myRank, int baseLine)


{
  for(int i = 1; i < numOfEntries-2; i++) {
       if (entries[i]->getValue() < baseLine) {
           cout << "Rank " << myRank
                  << " wrong answer i = " << i
                  << " baseLine = " << baseLine
                  << " value = " << entries[i]->getValue()
                 << endl;
           MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
       }



       if (*(entries[i]) > *(entries[i+1]) ) {
            cout << "Rank " << myRank
                 << " wrong answer i = " << i
                 << " value[i] = "
                 << entries[i]->getValue()
                 << " value[i+1] = "
                 << entries[i+1]->getValue()
                 << endl;
           MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
    }
}



//
//BlockOfEntries::printEntries
//
//Function: - Print myRank's entries to stdout.
//
void
BlockOfEntries::printEntries(int myRank)
{
    cout << endl;
    cout << "Rank " << myRank << endl;
    for(int i = 1; i < numOfEntries-1; i++)
          cout << entries[i]->getValue() << endl;
}



int
main(int argc, char **argv)



{
    int  myRank, numRanks;


MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD,&numRanks);



//
// Have each rank build its block of entries for the global sort.
```

```
//

   int numEntries;
   BlockOfEntries *aBlock = new BlockOfEntries(&numEntries,myRank);

//
// Compute the total number of entries and sort them.
//

       numEntries *= numRanks;
       for(int j = 0; j < numEntries / 2; j++) {


//
// Synchronize and then update the shadow entries.
//

       MPI_Barrier(MPI_COMM_WORLD);

       int recvVal, sendVal;
       MPI_Request sortRequest;
       MPI_Status status;

//
// Everyone except numRanks-1 posts a receive for the right's  rightShadow.
//

      if (myRank != (numRanks-1)) {
           MPI_Irecv(&recvVal, 1, MPI_INT, myRank+1,
                     MPI_ANY_TAG, MPI_COMM_WORLD,
                    &sortRequest);
      }



//
// Everyone except 0 sends its leftEnd to the left.
//


     if (myRank != 0) {
           sendVal = aBlock->getLeftEnd().getValue();
           MPI_Send(&sendVal, 1, MPI_INT,
                    myRank-1, 1, MPI_COMM_WORLD);
      }

      if (myRank != (numRanks-1)) {
           MPI_Wait(&sortRequest, &status);
                        aBlock->setRightShadow(Entry(recvVal));
      }


//
// Everyone except 0 posts for the left's leftShadow.
//

      if (myRank != 0) {
           MPI_Irecv(&recvVal, 1, MPI_INT, myRank-1,
                MPI_ANY_TAG, MPI_COMM_WORLD,
                &sortRequest);
      }


//
// Everyone except numRanks-1 sends its rightEnd right.
//

      if (myRank != (numRanks-1)) {
           sendVal = aBlock->getRightEnd().getValue();
           MPI_Send(&sendVal, 1, MPI_INT,
                    myRank+1, 1, MPI_COMM_WORLD);
      }
```

```
      if (myRank != 0) {
           MPI_Wait(&sortRequest, &status);
                   aBlock->setLeftShadow(Entry(recvVal));
      }


//
// Have each rank fix up its entries.
//

      aBlock->singleStepOddEntries();aBlock->singleStepEvenEntries();
   }


//
//  Print and verify the result.
//

   if (myRank == 0) {
      intsendVal;

      aBlock->printEntries(myRank);
      aBlock->verifyEntries(myRank, INT_MIN);

      sendVal = aBlock->getRightEnd().getValue();
      if (numRanks > 1)
           MPI_Send(&sendVal, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);
   } else {
      int          recvVal;
      MPI_Status  Status;
      MPI_Recv(&recvVal, 1, MPI_INT, myRank-1, 2,
                  MPI_COMM_WORLD, &Status);
      aBlock->printEntries(myRank);
      aBlock->verifyEntries(myRank, recvVal);
      if (myRank != numRanks-1) {
            recvVal = aBlock->getRightEnd().getValue();
                                    MPI_Send(&recvVal, 1,MPI_INT, myRank+1, 2,
MPI_COMM_WORLD);
      }
   }



   delete aBlock;
   MPI_Finalize();
   exit(0);
}
```

sort.C output

The output from running the sort executable is shown below. The application was run with -np4.

```
Rank 0
-998
-996
-996
-993

...
-567
-563
-544
-543

Rank 1
-535
-528
-528

...
-90
```

```
-90
-84
-84

Rank 2
-78
-70
-69
-69

...
383
383
386
386

Rank 3
386
393
393
397

...
950
965
987
987
```

compute_pi_spawn.f

This example computes pi by integrating f(x) = 4/(1 + x**2) using MPI_Spawn. It starts with one process and spawns a new world that does the computation along with the original process. Each newly spawned process receives the # of intervals used, calculates the areas of its rectangles, and synchronizes for a global summation. The original process 0 prints the result and the time it took.

```
  program mainprog
  include 'mpif.h'
  double precision PI25DT
  parameter(PI25DT = 3.14159265358979323846264338327950288419716939937510d0)
  double precision  mypi, pi, h, sum, x, f, a
  integer n, myid, numprocs, i, ierr
  integer parenticomm, spawnicomm, mergedcomm, high
C
C Function to integrate
C
  f(a) = 4.d0 / (1.d0 + a*a)
\
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  call MPI_COMM_GET_PARENT(parenticomm, ierr)
  if (parenticomm .eq. MPI_COMM_NULL) then
       print *, "Original Process ", myid, " of ", numprocs,
  +             " is alive"
      call MPI_COMM_SPAWN("./compute_pi_spawn", MPI_ARGV_NULL, 3,
  +              MPI_INFO_NULL, 0, MPI_COMM_WORLD, spawnicomm,
  +              MPI_ERRCODES_IGNORE, ierr)
      call MPI_INTERCOMM_MERGE(spawnicomm, 0, mergedcomm, ierr)
      call MPI_COMM_FREE(spawnicomm, ierr)
  else
       print *, "Spawned Process ", myid, " of ", numprocs,
  +             " is alive"
      call MPI_INTERCOMM_MERGE(parenticomm, 1, mergedcomm, ierr)
      call MPI_COMM_FREE(parenticomm, ierr)
  endif
  call MPI_COMM_RANK(mergedcomm, myid, ierr)
  call MPI_COMM_SIZE(mergedcomm, numprocs, ierr)
  print *, "Process ", myid, " of ", numprocs,
  +             " in merged comm is alive"
```

```
  sizetype = 1
  sumtype = 2
  if (myid .eq. 0) then
        n = 100
  endif
  call MPI_BCAST(n, 1, MPI_INTEGER, 0, mergedcomm, ierr)
C
C Calculate the interval size.
C
  h = 1.0d0 / n
  sum  = 0.0d0
  do 20 i = myid + 1, n, numprocs
      x = h * (dble(i) - 0.5d0)
      sum = sum + f(x)
20   continue
  mypi = h * sum
C
C Collect all the partial sums.
C
  call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
  +               MPI_SUM, 0, mergedcomm, ierr)
C
C Process 0 prints the result.
C
  if (myid .eq. 0) then
       write(6, 97) pi, abs(pi - PI25DT)
97           format('  pi is approximately: ', F18.16,
  +              '  Error is: ', F18.16)
  endif
  call MPI_COMM_FREE(mergedcomm, ierr)
  call MPI_FINALIZE(ierr)
  stop
  end
```

compute_pi_spawn.f output

The output from running the compute_pi_spawn executable is shown below. The application was run with -np1 and with the -spawn option.

```
Original Process  0 of  1 is alive
Spawned Process  0 of  3 is alive
Spawned Process  2 of  3 is alive
Spawned Process  1 of  3 is alive
Process  0 of  4 in merged comm is alive
Process  2 of  4 in merged comm is alive
Process  3 of  4 in merged comm is alive
Process  1 of  4 in merged comm is alive
pi is approximately: 3.1416009869231254
Error is: 0.0000083333333323
```

A P P E N D I X

B

Large message APIs

The current MPI standard allows the data transferred using standard API calls to be greater than 2 GB. For example, if you call MPI_Send()that contains a count of 1024 elements that each have a size of 2049 KB, the resulting message size in bytes is greater than what could be stored in a signed 32-bit integer.

Additionally, some users working with extremely large data sets on 64-bit architectures need to explicitly pass a count that is greater than the size of a 32-bit integer. The current MPI-2.1 standard does not accommodate this option. Until the standards committee releases a new API that does, Platform MPI provides new APIs to handle large message counts. These new APIs are extensions to the MPI-2.1 standard and will not be portable across other MPI implementations. These new APIs contain a trailing `L`. For example, to pass a 10 GB count to an MPI send operation, MPI_SendL()must be called, not MPI_Send ().

Important:

These interfaces will be deprecated when official APIs are included in the MPI standard.

The other API through which large integer counts can be passed into Platform MPI calls is the Fortran autodouble -i8interface (which is also nonstandard). This interface has been supported in previous Platform MPI releases, but historically had the limitation that the values passed in must still fit in 32-bit integers because the large integer input arguments were cast down to 32-bit values. For Platform MPI, that restriction is removed.

To enable Platform MPI support for these extensions to the MPI-2.1 standard, -non-standard-ext must be added to the command line of the Platform MPI compiler wrappers (`mpiCC`, `mpicc`, `mpif90`, `mpif77`), as in the following example:

```
% /opt/platform_mpi/bin/mpicc -non-standard-ext large_count_test.c
```

The -non-standard-ext flag must be passed to the compiler wrapper during the link step of building an executable.

The following is a complete list of large message interfaces supported.

Point-to-point communication

```
int MPI_BsendL(void *buf, MPI_Aint count, MPI_Datatype datatype,
int                          dest, int tag, MPI_Comm comm)
IN      buf            initial address of send buffer
```

```
IN      count          number of elements in send buffer
IN      datatype       datatype of each send buffer element
IN      dest           rank of destination
IN      tag            message tag
IN      comm           communicator
```

```
int MPI_Bsend_initL(void *buf, MPI_Aint count, MPI_Datatype datatype,
 int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN count
number                                  of elements sent (non-negative integer)
IN datatype type of each                                   element (handle) IN
dest rank of destination (integer) IN tag message
tag (integer) IN comm communicator (handle) OUT request
communication                                  request (handle)
```

```
int MPI_Buffer_attachL(void *buf, MPI_Aint size)
```

```
IN buffer initial buffer address (choice) IN size buffer size
in bytes
```

```
int MPI_Buffer_detachL(void *buf_address, MPI_Aint *size)
```

```
OUT buffer_addr initial buffer address (choice) OUT size
buffer size in bytes
```

```
int MPI_IbsendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                         dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements in send buffer IN datatype datatype of
each                                    send buffer element (handle) IN dest rank of
destination IN tag                                  message tag IN comm communicator
(handle) OUT request                                   communication request (handle)
```

```
int MPI_IrecvL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                         source, int tag, MPI_Comm comm, MPI_Request *request)
```

```
OUT buf initial address of receive buffer (choice) IN count number of elements
in receive buffer IN datatype datatype of each receive buffer element (handle)
IN source rank of source IN tag message tag IN comm communicator (handle) OUT
request communication request (handle)
```

```
int MPI_IrsendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                         dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN
count                                      number of elements in send buffer
IN datatype datatype of each                                        send buffer
element (handle) IN dest rank of destination IN
tag                                     message tag IN comm communicator
(handle) OUT request                                    communication request
(handle)
```

```
int MPI_IsendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                         dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN
count                                      number of elements in send buffer
IN datatype datatype of each                                        send buffer
element (handle) IN dest rank of destination IN
tag                                    message tag IN comm communicator OUT
request communication                                          request
```

```
int MPI_RecvL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                         source, int tag, MPI_Comm comm, MPI_Status *status)
```

```
OUT buf initial address of receive buffer (choice) IN
count number of elements in receive buffer IN datatype
datatype                                  of each receive buffer element (handle)
IN source rank of source                                  IN tag message tag IN comm
```

```
communicator (handle) OUT status                                    status object
(Status)
```

```
int MPI_Recv_initL(void* buf, MPI_Aint count, MPI_Datatype
datatype,                        int source, int tag, MPI_Comm comm, MPI_Request
*request)
```

```
OUT buf initial address of receive buffer (choice) IN
count number of elements received (non-negative integer)
IN                                  datatype type of each element (handle) IN source
rank of source                                  or MPI_ANY_SOURCE (integer) IN tag
message tag or MPI_ANY_TAG                                  (integer) IN comm
communicator (handle) OUT request                                   communication
request (handle)
```

```
int MPI_RsendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                          dest, int tag, MPI_Comm comm)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements in send buffer IN datatype datatype of
each                                    send buffer element (handle) IN dest rank of
destination IN tag                                  message tag IN comm communicator
(handle)
```

```
int MPI_Rsend_initL(void* buf, MPI_Aint count, MPI_Datatype
datatype,                        int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements sent IN datatype type of each element
(handle) IN dest rank of destination IN tag message tag IN
comm                                   communicator (handle) OUT request communication
request                                      (handle)
```

```
int MPI_SendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                          dest, int tag, MPI_Comm comm)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements in send buffer IN datatype datatype of
each                                    send buffer element (handle) IN dest rank of
destination IN tag                                  message tag IN comm communicator
(handle)
```

```
int MPI_Send_initL(void* buf, MPI_Aint count, MPI_Datatype
datatype,                        int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements sent IN datatype type of each element
(handle) IN dest rank of destination IN tag message tag IN
comm                                   communicator (handle) OUT request communication
request                                      (handle)
```

```
int MPI_SendrecvL(void *sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, int dest, int sendtag, void *recvbuf, MPI_Aint recvcount,
 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status
 *status)
```

```
IN sendbuf initial address of send buffer (choice) IN
sendcount number of elements in send buffer IN sendtype type
of                                     elements in send buffer (handle) IN dest rank
of destination IN                                   sendtag send tag OUT recvbuf
initial address of receive buffer                                   (choice) IN
recvcount number of elements in receive buffer IN
recvtype type of elements in receive buffer (handle) IN
source                                      rank of source IN recvtag receive tag IN
comm communicator                                    (handle) OUT status status object
(status)
```

```
int MPI_Sendrecv_replaceL(void* buf, MPI_Aint count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm
comm,                          MPI_Status *status)
```

```
INOUT buf initial address of send and receive buffer
(choice) IN count number of elements in send and receive
buffer                                     IN datatype type of elements in send and
receive buffer (handle)                                  IN dest rank of destination
IN sendtag send message tag IN                                  source rank of source
```

```
IN recvtag receive message tag IN comm                                    communicator
(handle) OUT status status object                                       (status)
```

```
int MPI_SsendL(void* buf, MPI_Aint count, MPI_Datatype datatype,
int                          dest, int tag, MPI_Comm comm)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements in send buffer IN datatype datatype of
each                                    send buffer element (handle) IN dest rank of
destination IN tag                                  message tag IN comm communicator
(handle)
```

```
int MPI_Ssend_initL(void* buf, MPI_Aint count, MPI_Datatype datatype,
 int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

```
IN buf initial address of send buffer (choice) IN count
number of elements sent IN datatype type of each element
(handle) IN dest rank of destination IN tag message tag IN
comm                                  communicator (handle) OUT request communication
request                                    (handle)
```

Collective communication

```
int MPI_AllgatherL(void* sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, void* recvbuf, MPI_Aint recvcount, MPI_Datatype recvtype, MPI_Comm
 comm)
```

```
IN sendbuf starting address of send buffer (choice)
IN                                      sendcount number of elements in send
buffer IN sendtype data                                          type of send buffer
elements (handle) OUT recvbuf address of
receive buffer (choice) IN recvcount number of elements
received                                      from any process IN recvtype data
type of receive buffer                                        elements (handle)
IN comm communicator (handle)
```

```
int MPI_AllgathervL(void* sendbuf, MPI_Aint sendcount,
MPI_Datatype                        sendtype, void* recvbuf, MPI_Aint *recvcounts, int
*displs, MPI_Datatype
recvtype, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) IN
sendcount number of elements in send buffer IN sendtype
data                                    type of send buffer elements (handle) OUT
recvbuf address of                                   receive buffer (choice) IN
recvcounts Array containing the                                 number of elements
that are received from each process IN displs                                 Array
of displacements relative to recvbuf IN recvtype data
type                                    of receive buffer elements (handle) IN comm
communicator                                   (handle)
```

```
int MPI_AllreduceL(void* sendbuf, void* recvbuf, MPI_Aint count,
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) OUT
recvbuf starting address of receive buffer (choice) IN
count                                  number of elements in send buffer IN datatype
data type of                                   elements of send buffer (handle) IN
op operation (handle) IN                                 comm communicator (handle)
```

```
int MPI_AlltoallL(void* sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, void* recvbuf, MPI_Aint recvcount, MPI_Datatype recvtype, MPI_Comm
 comm)
```

```
IN sendbuf starting address of send buffer (choice) IN
sendcount number of elements sent to each process IN
sendtype                                 data type of send buffer elements (handle)
OUT recvbuf address                                 of receive buffer (choice) IN
recvcount number of elements                                 received from any
process IN recvtype data type of receive                                    buffer
elements (handle) IN comm communicator                                    (handle)
```

```
int MPI_AlltoallvL(void* sendbuf, MPI_Aint *sendcounts, MPI_Aint
```

```
 *sdispls, MPI_Datatype sendtype, void* recvbuf, MPI_Aint *recvcounts,
 MPI_Aint *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) IN
sendcounts array equal to the group size specifying the
number                                    of elements to send to each rank IN sdispls
array of                                     displacements relative to sendbuf IN
sendtype data type of send                                      buffer elements (handle)
OUT recvbuf address of receive buffer                                        (choice) IN
recvcounts array equal to the group size specifying
the number of elements that can be received from each rank
IN                                  rdispls array of displacements relative to recvbuf
IN recvtype                                    data type of receive buffer elements
(handle) IN comm                                    communicator (handle)
```

```
int MPI_AlltoallwL(void *sendbuf, MPI_Aint sendcounts[], MPI_Aint
 sdispls[], MPI_Datatype sendtypes[], void *recvbuf, MPI_Aint recvcounts[],
 MPI_Aint rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) IN
sendcounts array equal to the group size specifying the
number                                    of elements to send to each rank IN sdispls
array of                                     displacements relative to sendbuf IN
sendtypes array of                                  datatypes, with entry j specifying
the type of data to send to                                   process j OUT recvbuf
address of receive buffer (choice) IN                                    recvcounts
array equal to the group size specifying the number
of elements that can be received from each rank IN rdispls
array                                     of displacements relative to recvbuf IN
recvtypes array of                                  datatypes, with entry j specifying
the type of data recieved                                  from process j IN comm
communicator (handle)
```

```
int MPI_BcastL(void* buffer, MPI_Aint count, MPI_Datatype datatype,
 int root, MPI_Comm comm ) INOUT buffer starting address of buffer
(choice)
```

IN count number of entries in buffer IN datatype data type of buffer (handle) IN root rank of broadcast root IN comm communicator (handle)

```
int MPI_GatherL(void* sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, void* recvbuf, MPI_Aint recvcount, MPI_Datatype recvtype, int
 root, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) IN
sendcount number of elements in send buffer IN sendtype
data                                    type of send buffer elements (handle) OUT
recvbuf address of                             receive buffer (choice, significant
only at root) IN recvcount                                  number of elements for
any single receive (significant only at                                    root) IN
recvtype data type of recv buffer elements (significant
only at root) (handle) IN root rank of receiving process
(integer) IN comm communicator (handle)
```

```
int MPI_GathervL(void* sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, void* recvbuf, MPI_Aint *recvcounts, MPI_Aint *displs,
 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice)
IN                                      sendcount number of elements IN send
buffer (non-negative                                    integer) IN sendtype
data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice, significant
only                                       at root) IN recvcounts array equal
to the group size specifying                                          the number
of elements that can be received from each rank
IN                                    displs array of displacements relative
to recvbuf IN recvtype                                      data type of recv
buffer elements (significant only at root)
(handle) IN root rank of receiving process (integer) IN
comm                                       communicator (handle)
```

```
int MPI_ReduceL(void* sendbuf, void* recvbuf, MPI_Aint count,
```

```
 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
IN sendbuf address of send buffer (choice) OUT recvbuf
address of receive buffer (choice, significant only at root)
IN                                  count number of elements in send buffer IN
datatype data type of                                elements of send buffer
(handle) IN op reduce operation (handle)                                  IN root
rank of root process IN comm communicator                                  (handle)
```

```
int MPI_Reduce_scatterL(void* sendbuf, void* recvbuf, MPI_Aint
*recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) OUT
recvbuf starting address of receive buffer (choice) IN
recvcounts array specifying the number of elements in
result                                    distributed to each process. IN datatype
data type of elements                                of input buffer (handle) IN
op operation (handle) IN comm                                 communicator (handle)
```

```
int MPI_ScanL(void* sendbuf, void* recvbuf, MPI_Aint count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

```
IN sendbuf starting address of send buffer (choice) OUT
recvbuf starting address of receive buffer (choice) IN
count                                  number of elements in input buffer IN datatype
data type of                                elements of input buffer (handle) IN
op operation (handle) IN                                   comm communicator (handle)
```

```
int MPI_ExscanL(void *sendbuf, void *recvbuf, MPI_Aint count,
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
IN sendbuf starting address of send buffer (choice) OUT
recvbuf starting address of receive buffer (choice) IN
count                                  number of elements in input buffer IN datatype
data type of                                elements of input buffer (handle) IN
op operation (handle) IN                                   comm intracommunicator
(handle)
```

```
int MPI_ScatterL(void* sendbuf, MPI_Aint sendcount, MPI_Datatype
 sendtype, void* recvbuf, MPI_Aint recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

```
IN sendbuf address of send buffer (choice, significant
only at root) IN sendcount number of elements sent to
each                                  process (significant only at root) IN sendtype
data type of send                           buffer elements (significant only
at root) (handle) OUT recvbuf                                  address of receive
buffer (choice) IN recvcount number of                                  elements in
receive buffer IN recvtype data type of receive
buffer elements (handle) IN root rank of sending process IN
comm                                   communicator (handle)
```

```
int MPI_ScattervL(void* sendbuf, MPI_Aint *sendcounts, MPI_Aint
 *displs, MPI_Datatype sendtype, void* recvbuf, MPI_Aint recvcount,
 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
IN sendbuf address of send buffer (choice, significant
only at root) IN sendcounts array specifying the number
of                                  elements to send to each processor IN displs
Array of                                 displacements relative to sendbuf IN
sendtype data type of send                                buffer elements (handle)
OUT recvbuf address of receive buffer                                  (choice) IN
recvcount number of elements in receive buffer IN
recvtype data type of receive buffer elements (handle) IN
root                                    rank of sending process IN comm
communicator                                   (handle)
```

Data types communication

```
int MPI_Get_countL(MPI_Status *status, MPI_Datatype datatype, MPI_Aint
```

```
 *count)

IN status return status of receive operation (status) IN
datatype datatype of each receive buffer entry (handle)
OUT                                    count number of received entries (integer)

int MPI_Get_elementsL(MPI_Status *status, MPI_Datatype datatype,
 MPI_Aint *count)

IN status return status of receive operation (status) IN
datatype datatype used by receive operation (handle) OUT
count                                     number of received basic elements (integer)

int MPI_PackL(void* inbuf, MPI_Aint incount, MPI_Datatype datatype,
 void *outbuf, MPI_Aint outsize, MPI_Aint *position, MPI_Comm
 comm)

IN inbuf input buffer start (choice) IN incount number
of input data items \ IN datatype datatype of each input
data                                      item (handle) OUT outbuf output buffer start
(choice) IN outsize                                      output buffer size, in bytes
INOUT position current position in                                          buffer in bytes
IN comm communicator for packed message                                           (handle)

int MPI_Pack_externalL(char *datarep, void *inbuf, MPI_Aint incount,
 MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint
 *position)

IN      datarep    data representation (string)
IN      inbuf      input buffer start (choice)
IN      incount    number of input data items
IN      datatype   datatype of each input data item (handle)
OUT     outbuf     output buffer start (choice)
IN      outsize    output buffer size, in bytes
INOUT   position   current position in buffer, in bytes

int MPI_Pack_sizeL(MPI_Aint incount, MPI_Datatype datatype,
MPI_Comm                          comm, MPI_Aint *size)

IN incount count argument to packing call IN datatype
datatype argument to packing call (handle) IN comm
communicator                                        argument to packing call (handle) OUT
size upper bound on size                                 of packed message, in bytes

int MPI_Pack_external_sizeL(char *datarep, MPI_Aint incount,
 MPI_Datatype datatype, MPI_Aint *size)

IN      datarep    data representation (string)
IN      incount    number of input data items
IN      datatype   datatype of each input data item (handle)
OUT     size       output buffer size, in bytes

int MPI_Type_indexedL(MPI_Aint count, MPI_Aint *array_of_blocklengths,
 MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype
 *newtype)

IN      count                  number of blocks
IN      array_of_blocklengths  number of elements per block
IN      array_of_displacements displacement for each block, in
                               multiples of oldtype extent
IN      oldtype                old datatype (handle)
OUT     newtype                new datatype (handle)

int MPI_Type_sizeL(MPI_Datatype datatype, MPI_Aint *size)

IN datatype datatype (handle) OUT size datatype size

int MPI_Type_structL(MPI_Aint count, MPI_Aint *array_of_blocklengths,
 MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype
 *newtype)

IN count number of blocks (integer) IN
array_of_blocklength number of elements in each block IN
array_of_displacements byte displacement of each block
IN                                     array_of_types type of elements in each block
```

```
(array of handles                                    to datatype objects) OUT newtype
new datatype                                      (handle)
```

```
int MPI_Type_vectorL(MPI_Aint count, MPI_Aint blocklength, MPI_Aint
                        stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
IN count number of blocks (nonnegative integer) IN
blocklength number of elements in each block IN stride number
of                                    elements between start of each block IN oldtype
old datatype                                    (handle) OUT newtype new datatype
(handle)
```

```
int MPI_UnpackL(void* inbuf, MPI_Aint insize, MPI_Aint *position, void
 *outbuf, MPI_Aint outcount, MPI_Datatype datatype, MPI_Comm
                        comm)
```

```
IN      inbuf        input buffer start (choice)
IN      insize       size of input buffer, in  bytes
INOUT   position     current position in bytes
OUT     outbuf       output buffer start (choice)
IN      outcount     number of items to be unpacked
IN      datatype     datatype of each output data item (handle)
IN      comm         communicator for packed message (handle)
```

```
int MPI_Unpack_externalL(char *datarep, void *inbuf, MPI_Aint insize,
 MPI_Aint *position, void *outbuf, MPI_Aint outcount, MPI_Datatype
 datatype)
```

```
IN      datarep    data representation (string
IN      inbuf      input buffer start (choice)
IN      insize     input buffer size, in  bytes
INOUT   position   current position in buffer, in bytes
OUT     outbuf     output buffer start (choice)
IN      outcount   number of output data items
IN      datatype   datatype of output data item (handle)
```

```
int MPI_Type_contiguousL(MPI_Aint count, MPI_Datatype oldtype,
 MPI_Datatype *newtype)
```

```
IN      count     replication count
IN      oldtype   old datatype (handle)
OUT     newtype   new datatype (handle)
```

```
int MPI_Type_create_hindexedL(MPI_Aint count, MPI_Aint
 array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype
 oldtype, MPI_Datatype *newtype)
```

```
IN      count                  number of blocks
IN      array_of_blocklengths  number of elements in each block
IN      array_of_displacements byte displacement of each block
IN      oldtype                old datatype
OUT     newtype                new datatype
```

```
int MPI_Type_create_hvectorL(MPI_Aint count, MPI_Aint blocklength,
 MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
IN      count             number of blocks
IN      blocklength       number of elements in each block
IN      stride            number of bytes between start of each block
IN      oldtype           old datatype (handle)
OUT     newtype           new datatype (handle)
```

```
int MPI_Type_create_indexed_blockL(MPI_Aint count, MPI_Aint
 blocklength, MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
 MPI_Datatype *newtype)
```

```
IN      count            length of array of displacements
IN      blocklength      size of block in array_of_displacements
                         array of displacements
IN      oldtype          old datatype (handle)
OUT     newtype          new datatype (handle)
```

```
int MPI_Type_create_structL(MPI_Aint count, MPI_Aint
 array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype
 array_of_types[], MPI_Datatype *newtype)
```

```
IN      count                  number of blocks
```

```
IN      array_of_blocklength   number of elements in each block
IN      array_of_displacements byte displacement of each block
IN      array_of_types         type of elements in each block
                               (array of handles to datatype objects)
OUT     newtype                new datatype (handle)
```

```
int MPI_Type_hindexedL(MPI_Aint count, MPI_Aint
 *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype
 oldtype, MPI_Datatype *newtype)
```

```
IN      count                  number of blocks
IN      array_of_blocklengths  number of elements in each block
IN      array_of_displacements byte displacement of each block
IN      oldtype                old datatype (handle)
OUT     newtype                new datatype (handle)
```

```
int MPI_Type_hvectorL(MPI_Aint count, MPI_Aint blocklength, MPI_Aint
 stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
IN      count           number of blocks
IN      blocklength     number of elements in  each block
IN      stride          number of bytes between start of each block
IN      oldtype         old datatype (handle)
OUT     newtype         new datatype (handle)
```

One-sided communication

```
int MPI_Win_createL(void *base, MPI_Aint size, MPI_Aint disp_unit,
 MPI_Info info, MPI_Comm comm, MPI_WIN *win)
```

```
IN      base          initial address of window (choice)
IN      size          size of window in  bytes
IN      disp_unit     local unit size for displacements, in  bytes
IN      info          info argument (handle)
IN      comm          communicator (handle)
OUT     win           window object returned by the call (handle)
```

```
int MPI_GetL(void *origin_addr, MPI_Aint origin_count, MPI_Datatype
 origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Aint
 target_count, MPI_Datatype target_datatype, MPI_WIN win)
```

```
OUT     origin_addr     initial address of origin   buffer (choice)
IN      origin_count    number of entries in origin  buffer
IN      origin_datatype datatype of each entry in origin buffer
                        (handle)
IN      target_rank     rank of target (nonnegative integer)
IN      target_disp     displacement from window start to the
                        beginning of the target buffer
IN      target_count    number of entries in  target buffer
IN      target_datatype datatype of each entry in target buffer (handle)
IN      win             window object used for communication (handle)
```

```
int MPI_PutL(void *origin_addr, MPI_Aint origin_count,
MPI_Datatype                        origin_datatype, int target_rank, MPI_Aint
target_disp, MPI_Aint
 target_count, MPI_Datatype target_datatype, MPI_WIN win)
```

```
IN      origin_addr      initial address of origin  buffer (choice)
IN      origin_count     number of entries in origin  buffer
IN      origin_datatype  datatype of each entry in origin buffer (handle)
IN      target_rank      rank of target
IN      target_disp      displacement from start of window to target buffer
IN      target_count     number of entries in target buffer
IN      target_datatype  datatype of each entry in target buffer (handle)
IN      win              window object used for communication (handle)
```

```
int MPI_AccumulateL(void *origin_addr, MPI_Aint origin_count,
 MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
 MPI_Aint target_count, MPI_Datatype target_datatype, MPI_Op op,
MPI_WIN                          win)
```

```
IN      origin_addr      initial address of buffer (choice)
IN      origin_count     number of entries in buffer
IN      origin_datatype  datatype of each buffer entry (handle)
```

```
IN      target_rank      rank of target
IN     target_disp     displacement from start of window to beginning of target buffer
IN      target_count     number of entries in target buffer
IN      target_datatype  datatype of each entry in target buffer (handle)
IN      op               reduce operation (handle)
IN      win              window object (handle)
```

A P P E N D I X

C

Standard Flexibility in Platform MPI

Platform MPI implementation of standard flexibility

Platform MPI contains a full MPI-2 standard implementation. There are items in the MPI standard for which the standard allows flexibility in implementation. This appendix identifies the Platform MPI implementation of many of these standard-flexible issues.

The following table displays references to sections in the MPI standard that identify flexibility in the implementation of an issue. Accompanying each reference is the Platform MPI implementation of that issue.

Table 21: Platform MPI implementation of standard-flexible issues

Reference in MPI Standard	The Platform MPI Implementation
MPI implementations are required to define the behavior of MPI_Abort (at least for a `comm` of *MPI_COMM_WORLD*). MPI implementations can ignore the `comm` argument and act as if `comm` was *MPI_COMM_WORLD*. See MPI-1.2 Section 7.5.	MPI_Abortkills the application. `comm` is ignored, and uses *MPI_COMM_WORLD*.
An implementation must document the implementation of different language bindings of the MPI interface if they are layered on top of each other. See MPI-1.2 Section 8.1.	Fortran is layered on top of C and profile entry points are given for both languages.
MPI does not mandate what an MPI process is. MPI does not specify the execution model for each process; a process can be sequential or multithreaded. See MPI-1.2 Section 2.6.	MPI processes are UNIX or Win32 console processes and can be multithreaded.

Reference in MPI Standard	The Platform MPI Implementation
MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their initial binding to physical processes. See MPI-1.2 Section 2.6.	Platform MPI provides the `mpirun -np #` utility and appfiles as well as start-up integrated with other job schedulers and launchers. See the relevant sections in this guide.
MPI does not mandate that an I/O service be provided, but does suggest behavior to ensure portability if it is provided. See MPI-1.2 Section 2.8.	Each process in Platform MPI applications can read and write input and output data to an external drive.
The value returned for *MPI_HOST* gets the rank of the host process in the group associated with *MPI_COMM_WORLD*. *MPI_PROC_NULL* is returned if there is no host. MPI does not specify what it means for a process to be a host, nor does it specify that a *HOST* exists.	Platform MPI sets the value of *MPI_HOST* to *MPI_PROC_NULL*.
MPI provides *MPI_GET_PROCESSOR_NAME* to return the name of the processor on which it was called at the moment of the call. See MPI-1.2 Section 7.1.1.	If you do not specify a host name to use, the host name returned is that of gethostname. If you specify a host name using the -h option to `mpirun`, Platform MPI returns that host name.
The current MPI definition does not require messages to carry data type information. Type information might be added to messages to allow the system to detect mismatches. See MPI-1.2 Section 3.3.2.	The default Platform MPI library does not carry this information due to overload, but the Platform MPI diagnostic library (DLIB) does. To link with the diagnostic library, use -ldmpi on the link line.
Vendors can write optimized collective routines matched to their architectures or a complete library of collective communication routines can be written using MPI point-to-point routines and a few auxiliary functions. See MPI-1.2 Section 4.1.	Use the Platform MPI collective routines instead of implementing your own with point-to-point routines. The Platform MPI collective routines are optimized to use shared memory where possible for performance.
Error handlers in MPI take as arguments the communicator in use and the error code to be returned by the MPI routine that raised the error. An error handler can also take stdargs arguments whose number and meaning is implementation dependent. See MPI-1.2 Section 7.2 and MPI-2.0 Section 4.12.6.	To ensure portability, the Platform MPI implementation does not take stdargs. For example in C, the user routine should be a C function of type MPI_handler_function, defined as:void (MPI_Handler_function) (MPI_Comm *, int *);
MPI implementors can place a barrier inside *MPI_FINALIZE*. See MPI-2.0 Section 3.2.2.	The Platform MPI *MPI_FINALIZE* behaves as a barrier function so that the return from *MPI_FINALIZE* is delayed until all potential future cancellations are processed.
MPI defines minimal requirements for thread-compliant MPI implementations and MPI can be implemented in environments where threads are not supported. See MPI-2.0 Section 8.7.	Platform MPI provides a thread-compliant library (`lmtmpi`), which only needs to be used for applications where multiple threads make MPI calls simultaneously (MPI_THREAD_MULTIPLE). Use -lmtmpi on the link line to use the `libmtmpi`.
The format for specifying the file name in *MPI_FILE_OPEN* is implementation dependent. An implementation might require that file name include a string specifying additional information about the file. See MPI-2.0 Section 9.2.1.	Platform MPI I/O supports a subset of the MPI-2 standard using ROMIO, a portable implementation developed at Argonne National Laboratory. No additional file information is necessary in your file name string.

A P P E N D I X

D

mpirun Using Implied prun or srun

Implied prun

Platform MPI provides an implied `prun` mode. The implied `prun` mode allows the user to omit the -prun argument from the `mpirun`command line with the use of the environment variable *MPI_USEPRUN*.

Set the environment variable:

```
% setenv MPI_USEPRUN 1
```

Platform MPI will insert the -prunargument.

The following arguments are considered to be `prun` arguments:

- -n -N -m -w -x
- -e MPI_WORKDIR=/path will be translated to the `prun` argument --chdir=/path
- any argument that starts with -- and is not followed by a space
- -np will be translated to -n
- -prun will be accepted without warning.

The implied `prun` mode allows the use of Platform MPI appfiles. Currently, an appfile must be homogenous in its arguments except for -h and -np. The -h and -np arguments in the appfile are discarded. All other arguments are promoted to the `mpirun` command line. Additionally, arguments following -- are also processed.

Additional environment variables provided:

- *MPI_PRUNOPTIONS*

  Allows additional `prun` options to be specified, such as --label.

  ```
  % setenv MPI_PRUNOPTIONS <option>
  ```

- *MPI_USEPRUN_IGNORE_ARGS*

  Provides an easy way to modify the arguments in an appfile by supplying a list of space-separated arguments that `mpirun` should ignore.

  ```
  % setenv MPI_USEPRUN_IGNORE_ARGS <option>
  ```

`prun` arguments:

- -n, --ntasks=ntasks

  Specify the number of processes to run.
- -N, --nodes=nnodes

  Request that nnodes nodes be allocated to this job.
- -m, --distribution=(block|cyclic)

  Specify an alternate distribution method for remote processes.
- -w, --nodelist=*host1*,*host2*,... or *file_name*

  Request a specific list of hosts.
- -x, --exclude=*host1*,*host2*,... or *file_name*

  Request that a specific list of hosts not be included in the resources allocated to this job.
- -l, --label

  Prepend task number to lines of stdout/err.

For more information on `prun` arguments, see the `prun` manpage.

Using the -prun argument from the `mpirun` command line is still supported.

Implied srun

Platform MPI provides an implied `srun` mode. The implied `srun` mode allows the user to omit the -srun argument from the `mpirun`command line with the use of the environment variable *MPI_USESRUN*.

Set the environment variable:

```
% setenv MPI_USESRUN 1
```

Platform MPI inserts the -srunargument.

The following arguments are considered to be `srun` arguments:

- -n -N -m -w -x
- any argument that starts with -- and is not followed by a space
- -np is translated to -n
- -srun is accepted without warning

The implied `srun` mode allows the use of Platform MPI appfiles. Currently, an appfile must be homogenous in its arguments except for -h and -np. The -h and -nparguments in the appfile are discarded. All other arguments are promoted to the `mpirun` command line. Additionally, arguments following -- are also processed.

Additional environment variables provided:

- *MPI_SRUNOPTIONS*

  Allows additional `srun` options to be specified such as --label.

  ```
  % setenv MPI_SRUNOPTIONS <option>
  ```
- *MPI_USESRUN_IGNORE_ARGS*

  Provides an easy way to modify arguments in an appfile by supplying a list of space-separated arguments that `mpirun` should ignore.

  ```
  % setenv MPI_USESRUN_IGNORE_ARGS <option>
  ```

In the example below, the appfile contains a reference to -stdio=bnone, which is filtered out because it is set in the ignore list.

% **setenv MPI_USESRUN_VERBOSE 1**

% **setenv MPI_USESRUN_IGNORE_ARGS -stdio=bnone**

% **setenv MPI_USESRUN 1**

% **setenv MPI_SRUNOPTION --label**

% **bsub -I -n4 -ext "SLURM[nodes=4]" $MPI_ROOT/bin/mpirun -stdio=bnone -f appfile -- pingpong**

```
Job <369848> is submitted to default queue <normal>.
<<Waiting for dispatch ...>>
<<Starting on lsfhost.localdomain>>
/opt/platform_mpi/bin/mpirun
unset
MPI_USESRUN;/opt/platform_mpi/bin/mpirun-srun ./pallas.x -npmin 4 pingpong
```

`srun` arguments:

- -n, --ntasks=ntasks

  Specify the number of processes to run.
- -N, --nodes=nnodes

  Request that nnodes nodes be allocated to this job.
- -m, --distribution=(block|cyclic)

  Specify an alternate distribution method for remote processes.
- -w, --nodelist=*host1*,*host2*,... or *filename*

  Request a specific list of hosts.
- -x, --exclude=*host1*,*host2*,... or *filename*

  Request that a specific list of hosts not be included in the resources allocated to this job.
- -l, --label

  Prepend task number to lines of stdout/err.

For more information on `srun` arguments, see the `srun`manpage.

The following is an example using the implied `srun` mode. The contents of the appfile are passed along except for -np and -hwhich are discarded. Some arguments are pulled from the appfile and others after the --.

Here is the appfile:

-np 1 -h foo -e MPI_FLAGS=T ./pallas.x -npmin 4

% **setenv MPI_SRUNOPTION "--label"**

These are required to use the new feature:

% **setenv MPI_USESRUN 1**

% **bsub -I -n4 $MPI_ROOT/bin/mpirun -f appfile -- sendrecv**

```
Job <2547> is submitted to default queue <normal>.
<<Waiting for dispatch ...>>
<<Starting on localhost>>
0: #---------------------------------------------------
0: #    PALLAS MPI Benchmark Suite V2.2, MPI-1 part
0: #---------------------------------------------------
0: # Date       : Thu Feb 24 14:24:56 2005
0: # Machine    : ia64# System : Linux
0: # Release    : 2.4.21-15.11hp.XCsmp
0: # Version    : #1 SMP Mon Oct 25 02:21:29 EDT 2004
0:
0: #
0: # Minimum message length in bytes:   0
0: # Maximum message length in bytes:   8388608
0: #
0: # MPI_Datatype                   :   MPI_BYTE
0: # MPI_Datatype for reductions    :   MPI_FLOAT
0: # MPI_Op                         :   MPI_SUM
0: #
0: #
0:
0: # List of Benchmarks to run:
0:
0: # Sendrecv
0:
0: #-----------------------------------------------------------------
0: # Benchmarking Sendrecv
0: # ( #processes = 4 )
0: #-----------------------------------------------------------------
0: #bytes #repetitions  t_min     t_max      t_avg    Mbytes/sec
0: 0        1000       35.28       35.40      35.34        0.00
0: 1        1000       42.40       42.43       42.41        0.04
0: 2        1000       41.60       41.69       41.64        0.09
0: 4        1000       41.82       41.91       41.86        0.18
0: 8        1000       41.46       41.49       41.48        0.37
```

```
0: 16       1000       41.19       41.27       41.21        0.74
0: 32       1000       41.44       41.54       41.51        1.47
0: 64       1000       42.08       42.17       42.12        2.89
0: 128      1000       42.60       42.70       42.64        5.72
0: 256      1000       45.05       45.08       45.07       10.83
0: 512      1000       47.74       47.84       47.79       20.41
0: 1024     1000       53.47       53.57       53.54       36.46
0: 2048     1000       74.50       74.59       74.55       52.37
0: 4096     1000      101.24      101.46      101.37       77.00
0: 8192     1000      165.85      166.11      166.00       94.06
0: 16384    1000      293.30      293.64      293.49      106.42
0: 32768    1000      714.84      715.38      715.05       87.37
0: 65536     640     1215.00     1216.45     1215.55      102.76
0: 131072    320     2397.04     2401.92     2399.05      104.08
0: 262144    160     4805.58     4826.59     4815.46      103.59
0: 524288     80     9978.35    10017.87     9996.31       99.82
0: 1048576    40    19612.90    19748.18    19680.29      101.28
0: 2097152    20    36719.25    37786.09    37253.01      105.86
0: 4194304    10    67806.51    67920.30    67873.05      117.79
0: 8388608     5   135050.20   135244.61   135159.04      118.30
0: #=================================================
0: #
0: # Thanks for using PMB2.2
0: #
0: # The Pallas team kindly requests that you
0: # give us as much feedback for PMB as possible.
0: #
0: # It would be very helpful when you sent the
0: # output tables of your run(s) of PMB to:
0: #
0: # pmb@pallas.com
0: #
0: # You might also add
0: #
0: # - personal information (institution, motivation
```

```
0: # for using PMB)
0: # - basic information about the machine you used
0: # (number of CPUs, processor type e.t.c.)
0: #
0: #=====================================================
0: MPI Rank        User (seconds)      System (seconds)
0:     0                     4.95                 2.36
0:     1                     5.16                 1.17
0:     2                     4.82                 2.43
0:     3                     5.20                 1.18
0:               ----------------      ----------------
0: Total:                   20.12                 7.13
```

`srun` is supported on HP XC systems with SLURM.

Using the -srunargument from the `mpirun` command line is still supported.

A P P E N D I X

E

Frequently Asked Questions

General

QUESTION: Where can I get the latest version of Platform MPI?

ANSWER: Customers can go to *my.platform.com*.

QUESTION: Can I use Platform MPI in my C++ application?

ANSWER: Yes, Platform MPI provides C++ classes for MPI bindings.The classes provided are an inlined interface class to MPI C bindings. Although most classes are inlined, a small portion is a prebuilt library. This library is g++ ABI compatible. Because some C++ compilers are not g++ ABI compatible, we provide the source files and instructions on how to build this library with your C++ compiler if necessary. For more information, see *C++ bindings (for Linux)* on page 54.

QUESTION: How can I tell what version of Platform MPI I'm using?

ANSWER: Try one of the following:

1. % **mpirun -version**
2. (on Linux) % **rpm -qa|grep "platform_mpi"**

For Windows, see the Windows FAQ section.

QUESTION: What Linux distributions does Platform MPI support?

ANSWER: See the release note for your product for this information. Generally, we test with the current distributions of RedHat and SuSE. Other versions might work, but are not tested and are not officially supported.

QUESTION: What is MPI_ROOT that I see referenced in the documentation?

ANSWER: *MPI_ROOT* is an environment variable that Platform MPI (`mpirun`) uses to determine where Platform MPI is installed and therefore which executables and libraries to use. It is especially helpful when you have multiple versions of Platform MPI installed on a system. A typical invocation of Platform MPI on systems with multiple *MPI_ROOT*s installed is:

% **setenv MPI_ROOT /scratch/test-platform-mpi-2.2.5/**

% **$MPI_ROOT/bin/mpirun ...**

Or

% **export MPI_ROOT=/scratch/test-platform-mpi-2.2.5**

% **$MPI_ROOT/bin/mpirun ...**

If you only have one copy of Platform MPI installed on the system and it is in `/opt/platform_mpi` or `/opt/mpi`, you do not need to set *MPI_ROOT*.

For Windows, see the Windows FAQ section.

QUESTION: Can you confirm that Platform MPI is include-file-compatible with MPICH?

ANSWER: Platform MPI can be used in what we refer to as MPICH compatibility mode. In general, object files built with the Platform MPI MPICH mode can be used in an MPICH application, and conversely object files built under MPICH can be linked into a Platform MPI application using MPICH mode. However, using MPICH compatibility mode to produce a single executable to run under both MPICH and Platform MPI is more problematic and is not recommended. For more information, see *MPICH object compatibility for Linux* on page 63.

Installation and setup

QUESTION: How are ranks launched? (Or, why do I get the message "remshd: Login incorrect" or "Permission denied"?)

ANSWER: There are a number of ways that Platform MPI can launch ranks, but some way must be made available:

1. Allow passwordless `rsh` access by setting up `hosts.equiv` and/or `.rhost` files to allow the `mpirun` machine to use `rsh` to access the execution nodes.
2. Allow passwordless `ssh` access from the `mpirun` machine to the execution nodes and set the environment variable *MPI_REMSH* to the full path of `ssh`.
3. Use SLURM (`srun`) by using the -srun option with `mpirun`.
4. Under Quadrics, use RMS (`prun`) by using the -prun option with `mpirun`.

For Windows, see the Windows FAQ section.

QUESTION: How can I verify that Platform MPI is installed and functioning optimally on my system?

ANSWER: A simple `hello_world` test is available in `$MPI_ROOT/help/hello_world.c` that can validate basic launching and connectivity. Other more involved tests are there as well, including a simple ping_pong_ring.c test to ensure that you are getting the bandwidth and latency you expect.

The Platform MPI for Linux library includes a lightweight system check API that does not require a separate license to use. This functionality allows customers to test the basic installation and setup of Platform MPI without the prerequisite of a license.

The `$MPI_ROOT/help/system_check.c`file contains an example of how this API can be used. This test can be built and run as follows:

% **$MPI_ROOT/bin/mpicc -o system_check.x $MPI_ROOT/help/system_check.c**

% **$MPI_ROOT/bin/mpirun ... system_check.x [***ppr_message_size***]**

Any valid options can be listed on the `mpirun`command line.

During the system check, the following tests are run:

1. `hello_world`
2. `ping_pong_ring`

These tests are similar to the code found in `$MPI_ROOT/help/hello_world.c` and `$MPI_ROOT/help/ping_pong_ring.c`. The `ping_pong_ring` test in `system_check.c` defaults to a message size

of 4096 bytes. An optional argument to the system check application can be used to specify an alternate message size. The environment variable *HPMPI_SYSTEM_CHECK* can be set to run a single test. Valid values of *HPMPI_SYSTEM_CHECK* are:

1. all: Runs both tests (the default value)
2. hw: Runs the `hello_world` test
3. ppr: Runs the `ping_pong_ring` test

If the *HPMPI_SYSTEM_CHECK* variable is set during an application run, that application runs normally until *MPI_Init* is called. Before returning from *MPI_Init*, the application runs the system check tests. When the system checks are completed, the application exits. This allows the normal application launch procedure to be used during the test, including any job schedulers, wrapper scripts, and local environment settings.

By default, the *HPMPI_SYSTEM_CHECK* API cannot be used if *MPI_Init* has already been called, and the API will call *MPI_Finalize* before returning.

QUESTION: Can I have multiple versions of Platform MPI installed and how can I switch between them?

ANSWER: You can install multiple Platform MPI's and they can be installed anywhere, as long as they are in the same place on each host you plan to run on. You can switch between them by setting *MPI_ROOT*. For more information on *MPI_ROOT*, refer to *General* on page 237.

QUESTION: How do I install in a non-standard location?

ANSWER: Two possibilities are:

% **rpm --prefix=*/whereverlyoulwant* -ivh pcmpi-*XXXXX.XXX*.rpm**

Or, you can basically use `untar` for an rpm using:

% **rpm2cpio pcmpi-*XXXXX.XXX*.rpm|cpio -id**

For Windows, see the Windows FAQ section.

QUESTION: How do I install a permanent license for Platform MPI?

ANSWER: You can install the permanent license on the server it was generated for by running **lmgrd -c *<full path to license file>***.

Building applications

QUESTION: Which compilers does Platform MPI work with?

ANSWER: Platform MPI works well with all compilers. We explicitly test with gcc, Intel, PathScale, and Portland. Platform MPI strives not to introduce compiler dependencies.

For Windows, see the Windows FAQ section.

QUESTION: What MPI libraries do I need to link with when I build?

ANSWER: We recommend using the `mpicc`, `mpif90`, and `mpi77` scripts in `$MPI_ROOT/bin` to build. If you do not want to build with these scripts, we recommend using them with the -show option to see what they are doing and use that as a starting point for doing your build. The -showoption prints out the command it uses to build with. Because these scripts are readable, you can examine them to understand what gets linked in and when.

For Windows, see the Windows FAQ section.

QUESTION: How do I build a 32-bit application on a 64-bit architecture?

ANSWER: On Linux, Platform MPI contains additional libraries in a 32-bit directory for 32-bit builds.

```
% $MPI_ROOT/lib/linux_ia32
```

Use the -mpi32 flag with `mpicc` to ensure that the 32-bit libraries are used. Your specific compiler might require a flag to indicate a 32-bit compilation.

For example:

On an Opteron system using gcc, you must instruct gcc to generate 32-bit via the flag -m32. The -mpi32 is used to ensure 32-bit libraries are selected.

```
% setenv MPI_ROOT /opt/platform_mpi
% setenv MPI_CC gcc
% $MPI_ROOT/bin/mpicc hello_world.c -mpi32 -m32
% file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2, dynamically linked (uses shared libraries),
not stripped
```

For more information on running 32-bit applications, see *Network specific* on page 241.

For Windows, see the Windows FAQ section.

Performance problems

QUESTION: How does Platform MPI clean up when something goes wrong?

ANSWER: Platform MPI uses several mechanisms to clean up job files. All processes in your application must call MPI_Finalize.

1. When a correct Platform MPI program (that is, one that calls MPI_Finalize) exits successfully, the root host deletes the job file.
2. If you use `mpirun`, it deletes the job file when the application terminates, whether successfully or not.
3. When an application calls MPI_Abort, MPI_Abort deletes the job file.
4. If you use `mpijob -j` to get more information on a job, and the processes of that job have exited, `mpijob` issues a warning that the job has completed, and deletes the job file.

QUESTION: My MPI application hangs at MPI_Send. Why?

ANSWER: Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode. Do not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. Platform MPI sometimes uses buffering for MPI_Send and MPI_Rsend, but it depends on message size and is at the discretion of the implementation.

QUESTION: How can I tell if the deadlock is because my code depends on buffering?

ANSWER: To quickly determine whether the problem is due to your code being dependent on buffering, set the z option for *MPI_FLAGS*. *MPI_FLAGS*modifies the general behavior of Platform MPI, and in this case converts MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without you needing to rewrite your code. MPI_Ssend guarantees synchronous send semantics, that is, a send can be started whether or not a matching receive is posted. However, the send completes successfully only if a matching receive is posted and the receive operation has begun receiving the message sent by the synchronous send.

If your application still hangs after you convert MPI_Send and MPI_Rsendcalls to MPI_Ssend, you know that your code is written to depend on buffering. Rewrite it so that MPI_Send and MPI_Rsend do not depend on buffering.

Alternatively, use non-blocking communication calls to initiate send operations. A non-blocking send-start call returns before the message is copied out of the send buffer, but a separate send-complete call is needed to complete the operation. For information about blocking and non-blocking communication, see *Sending and receiving messages* on page 17. For information about *MPI_FLAGS* options, see *General environment variables* on page 121.

QUESTION: How do I turn on MPI collection of message lengths? I want an overview of MPI message lengths being sent within the application.

ANSWER: The information is available through Platform MPI's instrumentation feature. Basically, including -i <*filename*> on the `mpirun` command line will create <*filename*> with a report that includes number and sizes of messages sent between ranks.

Network specific

QUESTION: I get an error when I run my 32-bit executable on my AMD64 or Intel(R)64 system.

```
dlopen for MPI_ICLIB_IBV__IBV_MAIN could not open libs in list libibverbs.so:
libibverbs.so: cannot open shared object file: No such file or directory
x: Rank 0:0: MPI_Init: ibv_resolve_entrypoints() failed
x: Rank 0:0: MPI_Init: Can't initialize RDMA device
x: Rank 0:0: MPI_Init: MPI BUG: Cannot initialize RDMA protocol dlopen for
MPI_ICLIB_IBV__IBV_MAIN could not open libs in list libibverbs.so:
libibverbs.so: cannot open shared object file: No such file or directory
x: Rank 0:1: MPI_Init: ibv_resolve_entrypoints() failed
x: Rank 0:1: MPI_Init: Can't initialize RDMA device
x: Rank 0:1: MPI_Init: MPI BUG: Cannot initialize RDMA protocol
MPI Application rank 0 exited before MPI_Init() with status 1
MPI Application rank 1 exited before MPI_Init() with status 1
```

ANSWER: Not all messages that say "Can't initialize RDMA device" are caused by this problem. This message can show up when running a 32-bit executable on a 64-bit Linux machine. The 64-bit daemon used by Platform MPI cannot determine the bitness of the executable and thereby uses incomplete information to determine the availability of high performance interconnects. To work around the problem, use flags (-TCP, -VAPI, etc.) to explicitly specify the network to use. Or, with Platform MPI 2.1.1 and later, use the -mpi32 flag to `mpirun`.

QUESTION: Where does Platform MPI look for the shared libraries for the high-performance networks it supports?

ANSWER: For information on high-performance networks, see *Interconnect support* on page 81.

QUESTION: How can I control which interconnect is used for running my application?

ANSWER: The environment variable *MPI_IC_ORDER* instructs Platform MPI to search in a specific order for the presence of an interconnect. The contents are a colon-separated list. For a list of default contents, see *Interconnect support* on page 81.

Or, `mpirun` command-line options can be used that take higher precedence than *MPI_IC_ORDER*. Lowercase selections imply to use if detected, otherwise keep searching. Uppercase selections demand the interconnect option be used, and if it cannot be selected the application terminates with an error. For a list of command-line options, see *Interconnect support* on page 81.

An additional issue is how to select a subnet when TCP/IP is used and multiple TCP/IP subnets are available between the nodes. This can be controlled by using the -netaddroption to `mpirun`. For example:

% mpirun -TCP -netaddr 192.168.1.1 -f appfile

This causes TCP/IP to be used over the subnet associated with the network interface with IP address 192.168.1.1.

For more detailed information and examples, see *Interconnect support* on page 81.

For Windows, see the Windows FAQ section.

Windows specific

QUESTION: What versions of Windows does Platform MPI support?

ANSWER: Platform MPI for Windows V1.0 supports Windows HPC. Platform MPI for Windows V1.1 supports Windows 2003 and Windows XP multinode runs with the Platform MPI Remote Launch service running on the nodes. This service is provided with V1.1. The service is not required to run in an SMP mode.

QUESTION: What is MPI_ROOT that I see referenced in the documentation?

ANSWER: *MPI_ROOT* is an environment variable that Platform MPI (`mpirun`) uses to determine where Platform MPI is installed and therefore which executables and libraries to use. It is especially helpful when you have multiple versions of Platform MPI installed on a system. A typical invocation of Platform MPI on systems with multiple *MPI_ROOT* variables installed is:

> set MPI_ROOT=\\nodex\share\test-platform-mpi-2.2.5

> "%MPI_ROOT%\bin\mpirun" ...

When Platform MPI is installed in Windows, it sets *MPI_ROOT* for the system to the default location. The default installation location differs between 32-bit and 64-bit Windows.

For 32-bit Windows, the default is:

```
C:\Program Files \Platform-MPI
```

For 64-bit Windows, the default is:

```
C:\Program Files (x86)\Platform-MPI
```

QUESTION: How are ranks launched on Windows?

ANSWER: On Windows HPC, ranks are launched by scheduling Platform MPI tasks to the existing job. These tasks are used to launch the remote ranks. Because CPUs must be available to schedule these tasks, the initial `mpirun` task submitted must only use a single task in the job allocation.

For additional options, see the release note for your specific version.

QUESTION: How do I install in a non-standard location on Windows?

ANSWER: To install Platform MPI on Windows, double-click setup.exe, and follow the instructions. One of the initial windows is the Select Directory window, which indicates where to install Platform MPI.

If you are installing using command-line flags, use **/DIR="<path>"** to change the default location.

QUESTION: Which compilers does Platform MPI for Windows work with?

ANSWER: Platform MPI works well with all compilers. We explicitly test with Visual Studio, Intel, and Portland compilers. Platform MPI strives not to introduce compiler dependencies.

QUESTION: What libraries do I need to link with when I build?

ANSWER: We recommend using the `mpicc` and `mpif90` scripts in `%MPI_ROOT%\bin` to build. If you do not want to build with these scripts, use them with the -show option to see what they are doing and use that as a starting point for doing your build.

The -show option prints out the command to be used for the build and not execute. Because these scripts are readable, you can examine them to understand what gets linked in and when.

If you are building a project using Visual Studio IDE, we recommend adding the provided `PMPI.vsprops` (for 32-bit applications) or `PMPI64.vsprops` (for 64-bit applications) to the property pages by using Visual Studio's Property Manager. Add this property page for each MPI project in your solution.

QUESTION: How do I specifically build a 32-bit application on a 64-bit architecture?

ANSWER: On Windows, open the appropriate compiler command window to get the correct 32-bit or 64-bit compilers. When using `mpicc` or `mpif90` scripts, include the -mpi32 or -mpi64 flag to link in the correct MPI libraries.

QUESTION: How can I control which interconnect is used for running my application?

ANSWER: The default protocol on Windows is TCP. Windows does not have automatic interconnect selection. To use InfiniBand, you have two choices: WSD or IBAL.

WSD uses the same protocol as TCP. You must select the relevant IP subnet, specifically the IPoIB subnet for InfiniBand drivers.

To select a subnet, use the -netaddr flag. For example:

```
R:\>mpirun -TCP -netaddr 192.168.1.1 -ccp -np 12 rank.exe
```

This forces TCP/IP to be used over the subnet associated with the network interface with the IP address 192.168.1.1.

To use the low-level InfiniBand protocol, use the -IBAL flag instead of -TCP. For example:

```
R:\> mpirun -IBAL -netaddr 192.168.1.1 -ccp -np 12 rank.exe
```

The use of -netaddr is not required when using -IBAL, but Platform MPI still uses this subnet for administration traffic. By default, it uses the TCP subnet available first in the binding order. This can be found and changed by going to the Network Connections > Advanced Settings windows.

IBAL is the desired protocol when using InfiniBand. IBAL performance for latency and bandwidth is considerably better than WSD.

For more information, see *Interconnect support* on page 81.

QUESTION: When I use 'mpirun -ccp -np 2 -nodex rank.exe' I only get one node, not two. Why?

ANSWER: When using the automatic job submittal feature of `mpirun`, -np *X* is used to request the number of CPUs for the scheduled job. This is usually equal to the number of ranks.

However, when using -nodex to indicate only one rank/node, the number of CPUs for the job is greater than the number of ranks. Because compute nodes can have different CPUs on each node, and `mpirun` cannot determine the number of CPUs required until the nodes are allocated to the job, the user must provide the total number of CPUs desired for the job. Then the -nodex flag limits the number of ranks scheduled to just one/node.

In other words, -np *X* is the number of CPUs for the job, and -nodex is telling `mpirun` to only use one CPU/node.

QUESTION: What is a UNC path?

ANSWER: A Universal Naming Convention (UNC) path is a path that is visible as a network share on all nodes. The basic format is:

\\node-name\exported-share-folder\paths

UNC paths are usually required because mapped drives might not be consistent from node to node, and many times don't get established for all logon tokens.

QUESTION: I am using mpirun automatic job submittal to schedule my job while in C:\tmp, but the job won't run. Why?

ANSWER: The automatic job submittal sets the current working directory for the job to the current directory (equivalent to using -e MPI_WORKDIR=<*path*>). Because the remote compute nodes cannot access local disks, they need a UNC path for the current directory.

Platform MPI can convert the local drive to a UNC path if the local drive is a mapped network drive. So running from the mapped drive instead of the local disk allows Platform MPI to set a working directory to a visible UNC path on remote nodes.

QUESTION: I run a batch script before my MPI job, but it fails. Why?

ANSWER: Batch files run in a command window. When the batch file starts, Windows first starts a command window and tries to set the directory to the 'working directory' indicated by the job. This is usually a UNC path so all remote nodes can see this directory. But command windows cannot change a directory to a UNC path.

One option is to use VBScript instead of .bat files for scripting tasks.

A P P E N D I X

F

Glossary

application

In the context of Platform MPI, an application is one or more executable programs that communicate with each other via MPI calls.

asynchronous

Communication in which sending and receiving processes place no constraints on each other in terms of completion. The communication operation between the two processes may also overlap with computation.

bandwidth

Data transmission capacity of a communications channel. The greater a channel's bandwidth, the more information it can carry per unit of time.

barrier

Collective operation used to synchronize the execution of processes. MPI_Barrier blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

blocking receive

Communication in which the receiving process does not return until its data buffer contains the data transferred by the sending process.

blocking send

Communication in which the sending process does not return until its associated data buffer is available for reuse. The data transferred can be copied directly into the matching receive buffer or a temporary system buffer.

broadcast

One-to-many collective operation where the root process sends a message to all other processes in the communicator including itself.

buffered send mode

Form of blocking send where the sending process returns when the message is buffered in application-supplied space or when the message is received.

buffering

Amount or act of copying that a system uses to avoid deadlocks. A large amount of buffering can adversely affect performance and make MPI applications less portable and predictable.

cluster

Group of computers linked together with an interconnect and software that functions collectively as a parallel machine.

collective communication

Communication that involves sending or receiving messages among a group of processes at the same time. The communication can be one-to-many, many-to-one, or many-to-many. The main collective routines are MPI_Bcast, MPI_Gather, and MPI_Scatter.

communicator

Global object that groups application processes together. Processes in a communicator can communicate with each other or with processes in another group. Conceptually, communicators define a communication context and a static group of processes within that context.

context

Internal abstraction used to define a safe communication space for processes. Within a communicator, context separates point-to-point and collective communications.

data-parallel model

Design model where data is partitioned and distributed to each process in an application. Operations are performed on each set of data in parallel and intermediate results are exchanged between processes until a problem is solved.

derived data types

User-defined structures that specify a sequence of basic data types and integer displacements for noncontiguous data. You create derived data types through the use of type-constructor functions that describe the layout of sets of primitive types in memory. Derived types may contain arrays as well as combinations of other primitive data types.

determinism

A behavior describing repeatability in observed parameters. The order of a set of events does not vary from run to run.

domain decomposition

Breaking down an MPI application's computational space into regular data structures such that all computation on these structures is identical and performed in parallel.

executable

A binary file containing a program (in machine language) which is ready to be executed (run).

explicit parallelism

Programming style that requires you to specify parallel constructs directly. Using the MPI library is an example of explicit parallelism.

functional decomposition

Breaking down an MPI application's computational space into separate tasks such that all computation on these tasks is performed in parallel.

gather

Many-to-one collective operation where each process (including the root) sends the contents of its send buffer to the root.

granularity

Measure of the work done between synchronization points. Fine-grained applications focus on execution at the instruction level of a program. Such applications are load balanced but suffer from a low computation/communication ratio. Coarse-grained applications focus on execution at the program level where multiple programs may be executed in parallel.

group

Set of tasks that can be used to organize MPI applications. Multiple groups are useful for solving problems in linear algebra and domain decomposition.

intercommunicators

Communicators that allow only processes in two different groups to exchange data.

intracommunicators

Communicators that allow processes within the same group to exchange data.

instrumentation

Cumulative statistical information collected and stored in ASCII format. Instrumentation is the recommended method for collecting profiling data.

latency

Time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

load balancing

Measure of how evenly the work load is distributed among an application's processes. When an application is perfectly balanced, all processes share the total work load and complete at the same time.

locality

Degree to which computations performed by a processor depend only upon local data. Locality is measured in several ways including the ratio of local to nonlocal data accesses.

locality domain (ldom)

Consists of a related collection of processors, memory, and peripheral resources that compose a fundamental building block of the system. All processors and peripheral devices in a given locality domain have equal latency to the memory contained within that locality domain.

mapped drive

In a network, drive mappings reference remote drives, and you have the option of assigning the letter of your choice. For example, on your local machine you might map S: to refer to drive C: on a server. Each time S: is referenced on the local machine, the drive on the server is substituted behind the scenes. The mapping may also be set up to refer only to a specific folder on the remote machine, not the entire drive.

message bin

A message bin stores messages according to message length. You can define a message bin by defining the byte range of the message to be stored in the bin: use the *MPI_INSTR* environment variable.

message-passing model

Model in which processes communicate with each other by sending and receiving messages. Applications based on message passing are nondeterministic by default. However, when one process sends two or more messages to another, the transfer is deterministic as the messages are always received in the order sent.

MIMD

Multiple instruction multiple data. Category of applications in which many instruction streams are applied concurrently to multiple data sets.

MPI

Message-passing interface. Set of library routines used to design scalable parallel applications. These routines provide a wide range of operations that include computation, communication, and synchronization. MPI-2 is the current standard supported by major vendors.

MPMD

Multiple data multiple program. Implementations of Platform MPI that use two or more separate executables to construct an application. This design style can be used to simplify the application source and reduce the size of spawned processes. Each process may run a different executable.

multilevel parallelism

Refers to multithreaded processes that call MPI routines to perform computations. This approach is beneficial for problems that can be decomposed into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to perform a computation and then joins after the computation is complete).

multihost

A mode of operation for an MPI application where a cluster is used to carry out a parallel application run.

nonblocking receive

Communication in which the receiving process returns before a message is stored in the receive buffer. Nonblocking receives are useful when communication and computation can be effectively overlapped in an MPI application. Use of nonblocking receives may also avoid system buffering and memory-to-memory copying.

nonblocking send

Communication in which the sending process returns before a message is stored in the send buffer. Nonblocking sends are useful when communication and computation can be effectively overlapped in an MPI application.

non-determinism

A behavior describing non-repeatable parameters. A property of computations which may have more than one result. The order of a set of events depends on run-time conditions and so varies from run to run.

OpenFabrics Alliance (OFA)

A not-for-profit organization dedicated to expanding and accelerating the adoption of Remote Direct Memory Access (RDMA) technologies for server and storage connectivity.

OpenFabrics Enterprise Distribution (OFED)

The open-source software stack developed by OFA that provides a unified solution for the two major RDMA fabric technologies: InfiniBand and iWARP (also known as RDMA over Ethernet).

parallel efficiency

An increase in speed in the execution of a parallel application.

point-to-point communication

Communication where data transfer involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model.

polling

Mechanism to handle asynchronous events by actively checking to determine if an event has occurred.

process

Address space together with a program counter, a set of registers, and a stack. Processes can be single threaded or multithreaded. Single-threaded processes can only perform one task at a time. Multithreaded processes can perform multiple tasks concurrently as when overlapping computation and communication.

race condition

Situation in which multiple processes vie for the same resource and receive it in an unpredictable manner. Race conditions can lead to cases where applications do not run correctly from one invocation to the next.

rank

Integer between zero and (number of processes - 1) that defines the order of a process in a communicator. Determining the rank of a process is important when solving problems where a master process partitions and distributes work to slave processes. The slaves perform some computation and return the result to the master as the solution.

ready send mode

Form of blocking send where the sending process cannot start until a matching receive is posted. The sending process returns immediately.

reduction

Binary operations (such as addition and multiplication) applied globally to all processes in a communicator. These operations are only valid on numeric data and are always associative but may or may not be commutative.

scalable

Ability to deliver an increase in application performance proportional to an increase in hardware resources (normally, adding more processors).

scatter

One-to-many operation where the root's send buffer is partitioned into *n* segments and distributed to all processes such that the *i*th process receives the *i*th segment. *n* represents the total number of processes in the communicator.

Security Support Provider Interface (SSPI)

A common interface between transport-level applications such as Microsoft Remote Procedure Call (RPC), and security providers such as Windows Distributed Security. SSPI allows a transport application to call one of several security providers to obtain an authenticated connection. These calls do not require extensive knowledge of the security protocol's details.

send modes

Point-to-point communication in which messages are passed using one of four different types of blocking sends. The four send modes include standard mode (MPI_Send), buffered mode (MPI_Bsend), synchronous mode (MPI_Ssend), and ready mode (MPI_Rsend). The modes are all invoked in a similar manner and all pass the same arguments.

shared memory model

Model in which each process can access a shared address space. Concurrent accesses to shared memory are controlled by synchronization primitives.

SIMD

Single instruction multiple data. Category of applications in which homogeneous processes execute the same instructions on their own data.

SMP

Symmetric multiprocessor. A multiprocess computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processes.

spin-yield

Refers to a Platform MPI facility that allows you to specify the number of milliseconds a process should block (spin) waiting for a message before yielding the CPU to another process. Specify a spin-yield value in the *MPI_FLAGS* environment variable.

SPMD

Single program multiple data. Implementations of Platform MPI where an application is completely contained in a single executable. SPMD applications begin with the invocation of a single process called the master. The master then spawns some number of identical child processes. The master and the children all run the same executable.

standard send mode

Form of blocking send where the sending process returns when the system can buffer the message or when the message is received.

stride

Constant amount of memory space between data elements where the elements are stored noncontiguously. Strided data are sent and received using derived data types.

synchronization

Bringing multiple processes to the same point in their execution before any can continue. For example, MPI_Barrier is a collective routine that blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

synchronous send mode

Form of blocking send where the sending process returns only if a matching receive is posted and the receiving process has started to receive the message.

tag

Integer label assigned to a message when it is sent. Message tags are one of the synchronization variables used to ensure that a message is delivered to the correct receiving process.

task

Uniquely addressable thread of execution.

thread

Smallest notion of execution in a process. All MPI processes have one or more threads. Multithreaded processes have one address space but each process thread contains its own counter, registers, and stack. This allows rapid context switching because threads require little or no memory management.

thread-compliant

An implementation where an MPI process may be multithreaded. If it is, each thread can issue MPI calls. However, the threads themselves are not separately addressable.

trace

Information collected during program execution that you can use to analyze your application. You can collect trace information and store it in a file for later use or analyze it directly when running your application interactively.

UNC

A Universal Naming Convention (UNC) path is a path that is visible as a network share on all nodes. The basic format is \\node-name\exported-share-folder\paths. UNC paths are usually required because mapped drives may not be consistent from node to node, and many times don't get established for all logon tokens.

yield

See spin-yield.

Index

/opt/mpi/doc 31
/opt/mpi/help 30
/opt/mpi/include 30
/opt/mpi/lib/pa2.0 31
/opt/mpi/newconfig/ 31

1sided option 109
32-bit applications 239
32-bit error 241
64-bit support 58

A

ADB 170
all-reduce 22
allgather 20
app bitness spec options 106
appfile
    adding program arguments 76
    assigning ranks in 76
    creating 75
    execution 73
    improving communication on multihost systems 76
    runs 70
    setting remote environment variables in 76
    with mpirun 67
appfile description of 71
application hangs 240
argument checking, enable 125
array partitioning 199
ASCII instrumentation profile 157
asynchronous communication 15
autodouble 56
    Linux 56
    Windows 56

B

backtrace 173
bandwidth 16, 163, 167
barrier 23, 167
binding ranks to ldoms 166
blocking communication 17
    buffered mode 18
    MPI_Bsend 18
    MPI_Recv 18
    MPI_Rsend 18
    MPI_Send 18
    MPI_Ssend 18
    read mode 18
    receive mode 17, 18
    send mode 17
    standard mode 18
    synchronous mode 18
blocking receive 18
broadcast 20, 21
buf variable 18–21
buffered send mode 17
build
    applications 66
    examples 184
        MPI
            on Linux cluster using appfiles 29
    MPI on HP XC cluster 30
    MPI on multiple hosts 75
    MPI on single host Linux 29
    MPI with Visual Studio 42
    problems with Windows 174
        run
            HPCS 39
            MPMD on HPCS 41
            multihost on HPCS 40
            single host on Windows 38
            Windows 2003/XP using appfiles 44
            Windows 2008 using appfiles 43
    Windows with Visual Studio 42

C

C bindings 237
C examples
    io.c 206
    ping_pong_ring.c 187
    ping_pong.c 185
    thread_safe.c 207
C++ 237
    bindings 54
    compilers 54
        examples
            cart.C 196
            sort.C 210
    profiling 159
cache option 115
cart.C 183
ccp option 113
ccpblock option 114
ccpcluster option 114
ccpcyclic option 114
ccperr option 113
ccpin option 113
ccpout option 114
ccpwait option 114
change execution location 126, 135
ck option 107
clean up 240
clearcache option 115
code
    a blocking receive 18
    a broadcast 21
    a nonblocking send 20
    a scatter 21
    error conditions 178
collective communication 20
    reduce 22
collective operations 20
    communication 20
    computation 22
    synchronization 23
comm variable 18–23
commd option 103
communication
    hot spots 77
    improving interhost 76
communicator
    determine number of processes 17
communicator.c 183
compilation utilities 31, 50
    Windows 46
compiler options
    -autodouble 56
    -i8 56
    -r16 56
    -r8 56
    -show 50
    +DD64 58
    32- and 64-bit library 58
compilers 239
    default 50
compiling
    applications 50
    Windows 36
completing Platform MPI 178
completion functions 113
completion routine 17
computation 22
compute_pi_spawn.f 184
compute_pi.f 183
configuration files 31
configure
    environment Linux 28
    Windows environment 35
connectx 108
constructor functions
    contiguous 24
    structure 24
    vector 24
context communication 19
context switching 165
contiguous and noncontiguous data 23
contiguous constructor 24
count variable 18–21
counter instrumentation 128, 156
cpu binding 60
cpu_bind 166
cpu_bind option 106
create
    appfile 75
    ASCII profile 156
    instrumentation profile 156

D

d option 107
daemons

multipurpose 78
number of processes 78
dbgspin option 108
dd option 108
DDE 121, 170
debug Platform MPI 121
debuggers 170
debugging
options 106
debugging Windows tutorial 173
default compilers 50
deferred deregistration 108
deregistration 150
derived data types 23
dest variable 18, 20
determine
group size 15
number of processes in communicator 17
rank of calling process 15
directory structure
Windows 45
download Platform MPI 237
dtype variable 18–21, 23
dump shmem configuration 125
dynamic processes 147

E

eadb 170
edde 121, 170
egdb 121, 170
elan 85
elan option 103
environment control options 109
environment variables
MPI_2BCOPY 127
MPI_BIND_MAP 126
MPI_CC 50
MPI_COMMD 131
MPI_CPU_AFFINITY 126
MPI_CPU_SPIN 126
MPI_CXX 50
MPI_DLIB_FLAGS 127
MPI_ERROR_LEVEL 128
MPI_F77 50
MPI_F90 50
MPI_FLAGS 121
MPI_FLUSH_FCACHE 126
MPI_GLOBMEMSIZE 136
MPI_IB_CARD_ORDER 134
MPI_IB_MULTIRAIL 131
MPI_IB_PKEY 134
MPI_IB_PORT_GID 132
MPI_IBV_QPPARAMS 135
MPI_IC_ORDER 130
MPI_IC_SUFFIXES 131
MPI_INSTR 128, 156
MPI_LOCALIP 138
MPI_MAX_REMSH 138
MPI_MAX_WINDOW 127
MPI_MT_FLAGS 125
MPI_NETADDR 138
MPI_NO_MALLOCLIB 136
MPI_NOBACKTRACE 128
MPI_PAGE_ALIGN_MEM 136
MPI_PHYSICAL_MEMORY 136
MPI_PIN_PERCENTAGE 137
MPI_RANKMEMSIZE 136
MPI_REMSH 138
MPI_ROOT 126
MPI_VAPI_QPPARAMS 135
MPI_WORKDIR 126
MPIRUN_OPTIONS 121
NLSPATH 154
setting in appfiles 76
setting in pcmpi.conf file 118
setting on Linux 118
setting on Windows 119
setting with command line 76, 118
TOTALVIEW 130
error checking, enable 125
error conditions 178
ewdb 121
example applications
cart.C 196
io.c 206
ping_pong_ring.c 187
exdb 121
external input and output 177

F

f option 105
failure detection 112
failure recover 111
file descriptor limit 177

Fortran 90 50
  examples master_worker.f90 195
functions MPI 57

G

gather 20
GDB 121, 170
getting started
  Linux 27
  Windows 27, 35
gm 85
gm option 102
gprof on HP XC 123

H

h option 106
ha option 109
header files 30, 45
headnode option 114
hello_world.c 184
help option 106
highly available infrastructure 110
hostfile option 105
hostlist option 106
hosts
  assigning using LSF 70
  multiple 75
  option 114

I

i option 107
I/O 229
ibal 83
ibal option 103
ibv 84
ibv option 102
implement barrier 23
implement reduction 22
implied
  prun 231
  prun mode 74
  srun 232
  srun mode 75
improve
  coding Platform MPI 167
  interhost communication 76
  network performance 164
InfiniBand
  card failover 84
  port failover 84
informational options 106
initialize MPI environment 15
installation 239
installation directory
  Linux 28
  Windows 35
instrumentation
  ASCII profile 157
  counter 156
  multihost 78
  output file 156
interconnects
  command-line options 81
  selection 81, 241
  selection examples 86
  selection options 102
  supported 6
  testing 179
interruptible collectives 112
intra=mix option 103
intra=nic option 103
intra=shm option 103
io.c 184
iscached option 115
itapi option 103

J

j option 107
job launcher options 104
job scheduler options 104
jobid option 114

L

language interoperability 122
large message APIs 219
latency 16, 163, 167
launch spec options 104
launching ranks 238
LD_LIBRARY_PATH appending 118
ldom 166
libraries to link 239

licenses 237
installing on Linux 33
installing on Windows 47
Linux 32
merging on Linux 33
release 149
testing on Linux 33
testing on Windows 47
Windows 45, 46
lightweight instrumentation 107, 111, 129
linking thread-compliant library 59
Linux
getting started 27
support 237
local host interconnect options 103
logical values in Fortran77 125
lsb_hosts option 104
lsb_mcpu_hosts option 104
LSF (load sharing facility) 70
LSF (load sharing facility) invoking 70, 75
LSF non-HP XC systems 70, 75
LSF on HP XC 69

M

manpages 31
compilation utilities
Windows 46
general
Windows 46
Linux 31
Platform MPI utilities 45
run-time 32
Windows 45, 46
master_worker.f90 183
messages
bandwidth
achieve highest 167
buffering problems 176
label 19
latency
achieve lowest 167
latency/bandwidth 162, 163
lengths 241
passing advantages 14
status 18
mode option 109
module.F 50
modules 72
MPI
allgather operation 20
alltoall operation 20
application
starting on Linux 28
broadcast operation 20
build application
on HP XC cluster 30
on Linux cluster using appfiles 29
on single host Linux 29
build application with Visual Studio 42
change execution source 126, 135
clean up 240
functions 57
gather operation 20
initialize environment 15
library routines
MPI_Comm_rank 15
MPI_Finalize 15
MPI_init 15
MPI_Recv 15
MPI_Send 15
number of 15
prefix 159
routine selection 167
run application
on HP XC cluster 30
on Linux cluster using appfiles 29
on single host Linux 29
run application Linux 28
run application on Elan interconnect 67
run application on Linux 67
scatter operation 20
terminate environment 15
MPI run application on multiple hosts 70
MPI_2BCOPY 127
MPI_Barrier 23, 167
MPI_Bcast 16, 21
MPI_BIND_MAP 61, 126
MPI_BOTTOM 122
MPI_Bsend 18
MPI_Cancel 123
MPI_Comm_disconnect 111
MPI_Comm_rank 15
MPI_COMMD 131
MPI_COPY_LIBHPC 143
MPI_CPU_AFFINITY 61, 126
MPI_CPU_SPIN 61, 126

MPI_DEBUG_CONT 170
MPI_DLIB_FLAGS 127
MPI_ERROR_LEVEL 128
MPI_FAIL_ON_TASK_FAILURE 143
MPI_Finalize 15
MPI_FLAGS 121, 162
MPI_FLUSH_FCACHE 61, 126
MPI_GLOBMEMSIZE 136
MPI_IB_CARD_ORDER 134
MPI_IB_MULTIRAIL 131
MPI_IB_PKEY 134
MPI_IB_PORT_GID 132
MPI_Ibsend 19
MPI_IBV_QPPARAMS 135
MPI_IC_ORDER 130
MPI_IC_SUFFIXES 131
MPI_Init 15
MPI_INSTR 128, 156
MPI_Irecv 19
MPI_Irsend 19
MPI_Isend 19, 20
MPI_Issend 19
MPI_LOCALIP 138
MPI_Lookup _name 153
MPI_MAX_REMSH 138
MPI_MAX_WINDOW 127
MPI_MT_FLAGS 125
MPI_NETADDR 138
MPI_NO_MALLOCLIB 136
MPI_NOBACKTRACE 128
MPI_NRANKS 144
MPI_PAGE_ALIGN_MEM 136
MPI_PHYSICAL_MEMORY 136
MPI_PIN_PERCENTAGE 137
MPI_PROT_BRIEF 141
MPI_PROT_MAX 141
MPI_PRUNOPTIONS 141
MPI_Publish _name 153
MPI_RANKID 144
MPI_RANKMEMSIZE 136
MPI_RDMA_INTRALEN 139
MPI_RDMA_MSGSIZE 139
MPI_RDMA_NENVELOPE 140
MPI_RDMA_NFRAGMENT 140
MPI_RDMA_NONESIDED 140
MPI_RDMA_NSRQRECV 140
MPI_Recv 15, 18
    high message bandwidth 167
    low message latency 167
MPI_Reduce 22
MPI_REMSH 138
MPI_ROOT 126, 237
MPI_Rsend 18
MPI_Rsend convert to MPI_Ssend 125
MPI_SAVE_TASK_OUTPUT 143
MPI_Scatter 21
MPI_Send 15, 18
    high message bandwidth 167
    low message latency 167
MPI_Send application hangs 240
MPI_Send convert to MPI_Ssend 125
MPI_SHMCNTL 125
MPI_SHMEMCNTL 137
MPI_SOCKBUFSIZE 142
MPI_SPAWN_SRUNOPTIONS 141
MPI_SRUNOPTIONS 141
MPI_Ssend 18
MPI_TCP_CORECVLIMIT 142
MPI_THREAD_AFFINITY 61
MPI_THREAD_IGNSELF 61
MPI_Unpublish _name 153
MPI_USE_LIBELAN 143
MPI_USE_LIBELAN_SUB 143
MPI_USE_MALLOPT_AVOID_MMAP 138
MPI_USEPRUN 141
MPI_USEPRUN_IGNORE_ARGS 142
MPI_USESRUN 142
MPI_VAPI_QPPARAMS 135
MPI_WORKDIR 126
MPI-2
    options 109
mpi32 option 106
mpi64 option 106
mpicc
    mpich 63
    on Windows 51
    utility 50
mpiCC
    utility 50
MPICH
    object compatibility 63
MPICH compatibility 238
MPICH2 compatibility 65
mpiclean 81, 178
mpidiag tool 95
mpiexec 78, 99

command-line options 79, 99
mpif77 utility 50
mpif90
on Windows 52
mpif90 utility 50
MPIHP_Trace_off 157
MPIHP_Trace_on 157
mpijob 79
mpirun 73, 95
appfiles 75
mpirun -version command 174
MPIRUN_OPTIONS 121
mpirun.mpich 63
mpiview file 156
MPMD
applications 71
with appfiles 72
with prun 72
with srun 72
multi_par.f 183
multilevel parallelism 24
multiple hosts 70
assigning ranks in appfiles 76
communication 76
multiple network interfaces 164
diagram of 164
improve performance 164
using 164
multiple threads 24, 167
multiple versions 239
mx option 102

N

name publishing 153
Native Language Support (NLS) 154
ndd option 108
netaddr option 104
network high availability 112
network interfaces 164
network selection options 102
NLS 154
NLSPATH 154
no clobber 129, 156
nodex option 114
nonblocking communication 17, 19
buffered mode 19
MPI_Ibsend 19
MPI_Irecv 19
MPI_Irsend 19
MPI_Isend 19
MPI_Issend 19
ready mode 19
receive mode 19
standard mode 19
synchronous mode 19
nonblocking send 20
noncontiguous and contiguous data 23
nonportable code
uncovering 125
nopass option 115
np option 106
number of MPI library routines 15

O

object compatibility 63
ofed 82, 84, 102
one-sided option 109
op variable 23
OPENMP
block partitioning 200
operating systems
supported 6
optimization report 124
option 109
options
password authentication 115
Windows 2003/XP 115
Windows CCP 113

P

p option 107
package option 115
packing and unpacking 23
parent process 20
pass option 115
PATH setting 28
pcmpi.conf 118
performance
communication hot spots 77
latency/bandwidth 162, 163
permanent license 239
ping_pong_clustertest.c 184
ping_pong_ring.c 179, 183
ping_pong.c 183

pk option 115
Platform MPI
    change behavior 121
    completing 178
    debug 169
    jobs running 79
    specify shared memory 136
    starting 73
    starting Linux 174
    utility files 45
platforms supported 6
PMPI
    prefix 159
point-to-point communications overview 16
portability 15
prefix
    for output file 156
    MPI 159
    PMPI 159
problems
    external input and output 177
    message buffering 176
    performance 162, 167
    run time 177
    shared memory 176
    with Windows build 174
process multithreaded 24
process placement multihost 77
process placement options 106
process rank of root 23
process rank of source 19
process single-threaded 24
processor
    locality 166
    subscription 165
profiling
    interface 159
progression 163
prot option 107
prun 74
    implied 231
    with mpirun 67
prun execution 74
prun MPI on Elan interconnect 67
prun option 105
psm option 102
pthreads 59
ptmalloc 150

R

rank of calling process 15
rank of source process 19
rank reordering 124
rdma option 108
RDMA options 108
ready send mode 17
receive buffer
    data type of elements 19
    number of elements in 19
    starting address 19
receive message
    information 19
    methods 17
recvbuf variable 22
recvcount variable 22
recvtype variable 22
reduce-scatter 22
reduction 22
reduction operation 23
release notes 31, 45
remote launch service 94
remote shell 70
    launching options 105
    remsh command 138, 174
    secure 28, 138
remsh 28, 70
reordering, rank 124
req variable 20
rhosts file 70, 174
root process 20
root variable 21–23
routine selection 167
rsh 28
run
    appfiles 70
    LSF on non-HP XC systems 70
        MPI
            on Linux cluster using appfiles 29
    MPI application 176
    MPI Linux application 67
    MPI on Elan interconnect 67
    MPI on HP XC 75
    MPI on HP XC cluster 30
    MPI on multiple hosts 75
    MPI on non-HP XC Linux 75
    MPI on single host Linux 29

MPI with appfiles 73
MPI with prun 74
MPI with srun 74
single host execution 73
Windows 36
with and appfile HPCS 44
run MPI on multiple hosts 70
run time
utilities
Windows 46
utility commands
mpiclean 81
mpijob 79
mpirun 73
run time environment variables 118
MPI_2BCOPY 127
MPI_BIND_MAP 126
MPI_COMMD 131
MPI_CPU_AFFINITY 126
MPI_CPU_SPIN 126
MPI_DLIB_FLAGS 127
MPI_ERROR_LEVEL 128
MPI_FLAGS 121
MPI_FLUSH_FCACHE 126
MPI_IB_CARD_ORDER 134
MPI_IB_MULTIRAIL 131
MPI_IB_PKEY 134
MPI_IB_PORT_GID 132
MPI_IBV_QPPARAMS 135
MPI_IC_ORDER 130
MPI_IC_SUFFIXES 131
MPI_INSTR 128
MPI_LOCALIP 138
MPI_MAX_REMSH 138
MPI_MAX_WINDOW 127
MPI_MT_FLAGS 125
MPI_NETADDR 138
MPI_NOBACKTRACE 128
MPI_RDMA_INTRALEN 139
MPI_RDMA_MSGSIZE 139
MPI_RDMA_NENVELOPE 140
MPI_REMSH 138
MPI_ROOT 126
MPI_SHMCNTL 125
MPI_SHMEMCNTL 137
MPI_USE_MALLOPT_AVOID_MMAP 138
MPI_VAPI_QPPARAMS 135
MPI_WORKDIR 126
MPIRUN_OPTIONS 121
run time utility commands 73, 95
run-time utilities 32

S

scalability 145
scan 22
scatter 20, 21
secure shell 28, 138
select reduction operation 22
send buffer data type of 23
send_receive.f 183
sendbuf variable 21
sendcount variable 21
sending data in one operation 16
sendtype variable 21
setting PATH 28
shared libraries 241
shared memory
control subdivision of 137
default settings 125
MPI_SHMEMCNTL 137
MPI_SOCKBUFSIZE 142
shared memory default settings 125
shell setting 28
signal propagation 151
single host execution 73
single-threaded processes 24
singleton launching 148
sort.C 184
source variable 19, 20
sp option 109
spawn 147
spawn option 109
spin/yield logic 123
SPMD 251
srq option 108
srun 74, 75
arguments 232, 233
examples 68
execution 74
implied 232
MPI_SRUNOPTIONS 141
option 105
with mpirun 68, 74
ssh 28, 138
standard send mode 17

starting
HP XC cluster applications 30
Linux cluster applications using appfiles 29
multihost applications 70
Platform MPI Linux 28, 174
Platform MPI Windows 175
singlehost applications on Linux 29
status 18
status variable 19
stdin 177
stdio 177, 229
stdio option 106
stdout 177
structure constructor 24
subdivision of shared memory 137, 139, 140
synchronization 23
synchronous send mode 17
system test 238

T

T option 108
tag variable 18–20
tcp interface options 104
TCP option 103
tcp/ip 83
terminate MPI environment 15
test system 238
thread multiple 24
thread_safe.c 184
thread-compliant library 59
+O3 59
+Oparallel 59
tk option 115
token option 115
TotalView 130
troubleshooting
Fortran 90 176
MPI_Finalize 178
Platform MPI 174
tunable parameters 162
tv option 108
twisted-data layout 200

U

udapl 85
udapl option 102
UNIX open file descriptors 177
unpacking and packing 23
using
counter instrumentation 156
multiple network interfaces 164
profiling interface 159

V

v option 107
vapi 85
vapi option 102
variables
buf 20
comm 20
count 20
dest 20
dtype 20
op 23
recvbuf 22, 23
recvcount 22
recvtype 22
req 20
root 23
sendbuf 21, 22
sendcount 21
sendtype 21
tag 20
vector constructor 24
version 34, 237
version option 107
viewing
ASCII profile 157

W

WDB 121
Windows
2003/XP command-line options 115
CCP command-line options 113, 115
getting started 27, 35

X

XDB 121, 170
xrc 108
xrc option 108

Y

yield/spin logic 123

zero-buffering 125

Z